Jalios JCMS 9 – Documentation Développeur

API de gestion des données

Introduction

L'API de gestion des données est au cœur du fonctionnement de JCMS. JCMS repose sur un système hybride de persistance des données dans lequel plusieurs systèmes de stockage sont utilisés. L'API de gestion des données fournit les interfaces pour accéder aux données quel que soit leur mode de stockage.

Ce document décrit les principes de stockage des données dans JCMS, le modèle de données et l'accès aux données en lecture comme en écriture. Les droits d'accès, le contrôle des écritures et la gestion des fichiers sont aussi abordés.

Contenu

1 Le stockage des données dans JCMS

1.1 Un stockage hybride

JCMS repose sur un stockage des données hybride. JCMS utilise plusieurs systèmes de stockage afin d'exploiter au mieux leurs propriétés et leurs spécificités.

Les principaux systèmes de stockage utilisés par JCMS sont :

- JStore : une base objet en mémoire dans la mouvance NoSQL. JCMS y stocke essentiellement les données de structure, les comptes utilisateurs et les contenus éditoriaux. JStore est couplé avec JSync qui assure la haute disponibilité des données.
- JcmsDB : une base de données relationnelle « classique » qui fonctionne sur la plupart des SGBDR du marché. JCMS y stocke les contenus utilisateurs, les données à forte volumétrie ou fréquemment mises à jour.
- Lucene : JCMS utilise le moteur de recherche Lucene pour indexer les données textuelles de JStore et de JcmsDB
- Système de fichier : JCMS stocke sur le système de fichier les documents déposés, les fichiers générés (vignettes, PDF, SWF, …), les propriétés du site, les types de publication et les types de workflow, les logs, ...

1.2 JStore

1.2.1 Fonctionnement général

JStore est une base de données objet en mémoire et dont la persistance est assurée par la journalisation des opérations de modification.

Lorsqu'une donnée, représentée par un objet Java, est créée, modifiée ou supprimée, l'opération décrivant cette écriture est ajoutée au journal. A chaque opération, seuls les attributs concernés sont enregistrés. Par exemple, lors de la mise à jour du titre d'un article, l'opération ne porte que sur cet attribut. Les opérations sont enregistrées dans le fichier store.xml. Chaque opération est représentée par une balise XML. Le nom de la balise représente la classe d'objet concernée (p. ex. member pour les membres, category pour les catégories, …). En plus des attributs propres à l'écriture, chaque opération comporte 3 attributs systématiquement présents :

- `stamp` : une estampille logique permettant d'identifier et d'ordonner cette opération de façon unique dans le cluster ;
- `id` : l'identifiant de l'objet sur lequel porte l'opération ;

- `op` : type d'opération (« create », « update », « delete »)
- `mdate` : la date à laquelle a eu lieu l'opération.

Exemple (simplifié) :

```
<member stamp="c_50001" id="c_50001" op="create" cdate="1377609854772" firstName="John"
login="jd" mdate="1377609854772" name="Doe" opAuthor="j_2" password="$2a$10$" />

<member stamp="c_50002" id="c_50001" op="update" email="john.doe@example.com"
mdate="1377609870256"  />

<group stamp="c_50003" id="c_50003" op="create" cdate="1377609887673" mdate="1377609887673"
name="Customers" />

<member stamp="c_50004" id="c_50001" op="update" declaredGroups="@|c_50003"
mdate="1377609905788"  />

<member stamp="c_50005" id="c_50001" op="delete" mdate="1377609912660" />
```

Dans l'exemple ci-dessus :

- L'opération c_50001 représente la création du membre John Doe,
- L'opération c_50002 représente la mise à jour de son email
- L'opération c_50003 représente la création du groupe Customers
- L'opération c_50004 représente le rattachement de John Doe au groupe Customers
- L'opération c_50005 représente la suppression du membre John Doe.

JStore maintient la cohérence entre l'état mémoire des objets et la journalisation des opérations d'écriture amenant à cet état mémoire.

La structure du store permet d'accéder à l'ensemble des versions d'une donnée ainsi qu'aux données supprimées. Afin d'accélérer l'accès à ces informations, JStore maintient un index des opérations de création. Ainsi, pour retrouver l'historique d'une donnée, JStore ne relit pas tout le journal mais se positionne directement à l'opération de création de la donnée. Une fois l'historique d'une donnée chargé il est placé dans un cache de type LRU (*Least Recently Used*).

1.2.2 Chargement du store

Au démarrage de l'application, l'ensemble des objets est chargé en mémoire en rejouant les opérations contenues dans le fichier store.xml.

Seuls les attributs des objets sont chargés en mémoire et stockés dans store.xml. Cela ne concerne donc pas les fichiers déposés sur le serveur, les gabarits de présentation (JSP), les feuilles de styles, …

JStore rétablit aussi tous les liens entre les objets. Il est ainsi possible de parcourir très simplement le graphe des objets. Par exemple pour accéder à partir d'une publication à la liste des administrateurs de l'espace de travail auquel elle appartient, il suffit d'écrire :

```
pub.getWorkspace().getAdministrators();
```

JCMS maintient de nombreux index sur les objets afin d'obtenir rapidement l'ensemble des instances d'une classe. Ces index sont maintenus en mémoire et sont donc très performant. JCMS maintient ainsi l'ensemble des membres d'un espace de travail, l'ensemble des membres d'un groupe, l'ensemble des publications d'un membre, l'ensemble des publications d'une catégorie, …

Deux métriques sont donc à considérer avec JStore : le volume mémoire et le temps de chargement.

1.2.3 Volumétrie

Le volume mémoire dépend du nombre d'objets vivants à charger et de la taille moyenne de ces objets. La taille d'un objet est essentiellement liée au volume de texte stocké dans l'objet (typiquement, un objet JCMS sans aucun champ texte pèse moins de 1 Ko). JStore repose sur les API Java dans lesquelles les chaînes de caractères sont stockées en Unicode sur 2 octets. Le volume mémoire occupé est donc un peu plus du double du volume de texte. Par exemple, une page A4 de texte représentant 3 Ko de texte brut sur disque occupera 7 Ko en mémoire dans JStore (index compris). Il faut ajouter à cela le poids des métadonnées associé aux contenus (de l'ordre de 1 Ko par contenu) ainsi que le poids des autres données présentent. Un site JCMS gérant 100 000 contenus de ce type occupera de l'ordre de 1,2 Go en mémoire.

En règle générale, on peut considérer les métriques suivantes :

- Les petits objets (membres, groupes, catégories, portlets, …) occupent en moyenne 3 Ko en mémoire. 300 000 objets de cette sorte occupent environ 1 Go de mémoire
- Les gros objets (contenus avec des champs textes) occupent en moyenne 10 Ko en mémoire. 100 000 objets de cette sorte occupent environ 1 Go de mémoire.

1.2.4 Temps de chargement

Le temps de chargement dépend du nombre d'opérations présentes dans le journal store.xml. Ce sont essentiellement les opérations de création qui sont coûteuses. Le temps de chargement est proportionnel au nombre et à la taille des objets. Par exemple, un journal de 100 000 objets représentant chacun une page A4 de texte brut (3 Ko sur disque) se charge en 21 secondes sur un PC Portable Dell Latitude E6520 avec un processeur i7-2720QM à 2,2 GHz avec 16 Go de RAM (dont 2 Go alloué à la JVM) et un disque Samsung SSD 840 Pro.

1.2.5 Nettoyage et compactage du store

Chaque écriture entrainant un ajout dans le fichier store.xml, celui-ci grossit proportionnellement au taux d'écriture. Un store volumineux diminue les performances du temps de chargement au démarrage de JCMS, de l'accès aux versions et de certaines synchronisations JSync.

Le nettoyage (ou compactage) du store consiste à diminuer le poids du fichier store.xml en regroupant ou en supprimant certaines opérations.

Le compactage est recommandé principalement dans deux cas :

1. Les temps de chargement du store ou d'accès aux versions sont prohibitifs.
2. Lorsqu'un store de développement doit être fusionné avec un store de production, il est recommandé de compacter le suffixe de développement. Ceci permet d'éliminer des données transitoires (p. ex. des données de test) et de simplifier le suffixe à fusionner.

Le compactage agit sur des séries d'opérations de mise à jour (update) sur un même objet et sur toutes les opérations portant sur des objets détruits.

En supprimant ces opérations, le compactage détruit une partie des versions des objets. Aussi, il est possible d'appliquer des règles pour choisir les opérations qui peuvent être compactées. Par

exemple, ne pas compacter les opérations portant sur les métadonnées d'une publication ou celles portant sur des publications appartenant à certains espaces de travail, …

Le compactage est une opération d'exploitation sensible. Elle nécessite un arrêt des écritures dans le cluster et un redémarrage de chaque instance.

1.2.6 Fusion de store avec le StoreMerge

Il est parfois nécessaire de fusionner les opérations de deux Stores. C'est par exemple le cas lors des mises en production. Il faut fusionner les opérations sur les données réalisées durant la période de développement et celles qui ont eu lieu en production durant cette période.

Le StoreMerge est un outil permettant de fusionner deux stores. Il identifie la partie commune et fusionne les deux suffixes divergents. Durant cette fusion les conflits sont détectés (p. ex. mise à jour concurrente de la même donnée). Des règles de résolution peuvent être fournies au StoreMerge pour résoudre les conflits.

L'article *StoreMerge - Guide de l'utilisateur* (http://community.jalios.com/howto/storemerge) détaille le fonctionnement et l'utilisation du StoreMerge.

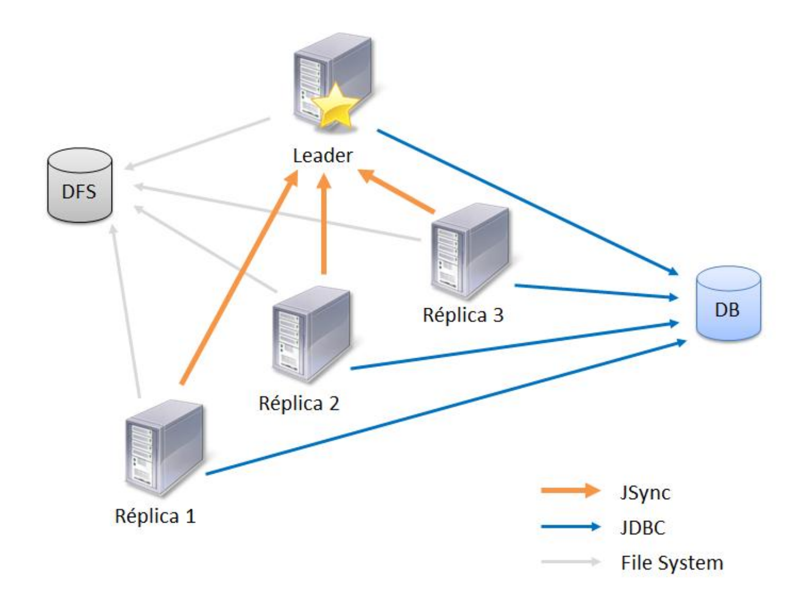
1.2.7 Réplication avec JSync

Afin de garantir la haute disponibilité d'un site JCMS, il est possible de répartir la charge sur plusieurs webapps via un répartiteur de charge. On parle alors d'un cluster JCMS. Chaque webapp ayant sa propre gestion mémoire il faut garantir que les données créées dans le store d'une webapp seront bien diffusées aux autres webapps du cluster.

JSync est le protocole de réplication de JStore permettant de construire un cluster JCMS. Dans ce cluster, chaque nœud est une instance de JCMS appelée un réplica. Les réplicas sont organisés en groupe avec un leader. Les réplicas se connectent à leur leader. C'est ce ralliement qui fait qu'un réplica devient le leader du groupe. Un leader peut lui-même être connecté avec un le leader d'un autre groupe.

Chaque instance JCMS contient l'ensemble des données du store mais l'ensemble du cluster doit se connecter à la même base de données JcmsDB. Pour assurer la haute disponibilité cette base peut elle-même être montée en cluster.

Pour les fichiers déposés, il est possible soit de les répliquer sur chaque nœud du cluster soit de les centraliser (p. ex. sur un SAN).



Chaque nœud accepte les nouvelles écritures (création, modification, suppression) et les envoie à son leader. Celui-ci les propage aux autres réplicas du groupe assurant ainsi la convergence globale du groupe.

JSync est un protocole de réplication optimiste de type *peer-to-peer*. JSync relâche la cohérence du système afin de garantir sa disponibilité et supporter des partitionnements (le « A » et le « P » du [théorème CAP](). JStore repose sur le principe de cohérence à terme (*eventual consistency*). Si on arrête les écritures dans le cluster et que l'on propage les écritures à l'ensemble des réplicas, alors le cluster est cohérent.

Un réplica peut se déconnecter, diverger puis se reconnecter pour diffuser ses nouveautés.

JCMS propose deux modes de propagation des changements :

- propagation automatique dès que les nouvelles écritures se sont stabilisées ;
- propagation à la demande (de l'administrateur ou via l'API JSync)

Durant les phases de divergences, des conflits sur les données peuvent apparaître. Par exemple, une même donnée peut être modifiée sur un réplica et détruite sur un autre. Lors de la phase de diffusion des nouveautés, JSync résout automatiquement les conflits d'écriture portant sur une même donnée. Les suppressions l'emportent sur les mises à jour. Les mises à jour concurrentes sont réordonnancées de façon cohérente dans l'ensemble du cluster. Les conflits inter-objets ne sont pas traités (par exemple, suppression d'une catégorie sur un réplica et ajout d'un contenu référençant cette catégorie sur un autre réplica).

Afin de réduire les risques de conflits, il est recommandé de réduire les périodes de divergence. Idéalement, les contributeurs agissant sur des données liées doivent être regroupés sur un même réplica ou sur des réplicas configurés en propagation immédiate. Les lecteurs peuvent par contre être répartis sur l'ensemble des réplicas du groupe. Le regroupement des contributeurs peut être fait explicitement (p. ex. avec une URL dédiée à la contribution) ou automatiquement avec le module de centralisation des écritures (http://community.jalios.com/plugin/centralizedwrites).

L’article *Mise en œuvre d'un site JCMS à haute disponibilité avec le protocole de réplication JSync* (http://community.jalios.com/howto/jsync) détaille la mise en place de JSync.

1.3 JcmsDB

JCMS stocke une partie de ses données dans une base de données relationnelle, JcmsDB, gérée dans un SGBDR.

JCMS est livré avec le SGBDR embarqué Derby. Ce SGBDR est bien adapté pour les environnements de développement et les sites faisant un usage très modéré de JcmsDB. Au-delà, il est recommandé d’utiliser l’un des SGBDR externes certifié pour JCMS (MySQL, PostgreSQL, Oracle, SQLServer, DB2) Ce choix peut aussi être motivé si vous souhaitez disposer d’outils d’administration et d’exploitation du SGBDR. Attention ! JCMS n’assure pas la migration de données d’un SGBDR à un autre. Aussi, le choix du SGBDR doit être mûrement réfléchi dès la phase d’architecture du projet.

Si vous retenez Derby, il n’y a aucun paramétrage à faire. JCMS se charge de la sauvegarde quotidienne de la base. Dans le cas d’un SGBDR externe, la création de la base et les procédures d’exploitation (notamment la sauvegarde régulière) sont à la charge de l’administrateur du SGBDR.

1.3.1 Hibernate

JCMS utilise le framework de mapping objet/relationnel (ORM) Hibernate pour gérer les données dans JcmsDB. Grâce à son système de dialectes, Hibernate offre à la fois une grande indépendance du code vis-à-vis de la base sous-jacente et de bonnes performances. Hibernate supporte la grande majorité des SGBDR. Néanmoins, en pratique, il reste certaines spécificités propres à chaque SGBDR qu'il faut prendre en compte (p. ex. mots-clés réservés, longueurs des noms des tables, unicité des index, gestion des CLOB, ...). Aussi, afin de garantir un bon fonctionnement, JCMS a été certifié sur certains d'entre eux. Reportez-vous au manuel d’installation et d’exploitation de votre version de JCMS pour avoir la liste précise de SGBDR supporté.

Au premier démarrage, JCMS utilise Hibernate pour générer dans JcmsDB l’ensemble des tables et des index nécessaires. Cette structure peut évoluer selon les créations et modifications des types de publications et les ajouts de modules.

1.3.2 Transaction

JcmsDB fonctionne en mode transactionnel. Tous les accès aux données de JcmsDB, aussi bien en lecture qu’en écriture, doivent être encapsulés dans une transaction.

Pour tous les accès Web, JCMS prend en charge l’ouverture et la fermeture de la transaction à chaque requête (via la ServletFilter HibernateSessionFilter).

Pour les accès lors du déclenchement d’une alarme, JCMS fournit la classe TransactionalAlarmListener qui encapsule l’appel au code qui prend en charge l’alarme.

Pour tous les autres cas d’usage, le développeur doit gérer lui-même la transaction. C’est notamment le cas dans les tests unitaires ou lors de l’accès à JcmsDB depuis un thread spécifique (p. ex. dans un cas de producteur/consommateur).

1.3.3 JcmsDB et JSync

Le protocole de réplication JSync n'opère que sur les données gérées dans JStore. Il ne traite donc pas les données stockées dans JcmsDB.

Lorsqu'une écriture a lieu dans JcmsDB des traitements consécutifs à cette écriture sont déclenchés (p. ex. indexation de la donnée, invalidation de cache, …) Ces traitements sont pris en charge par des classes Java implémentant l'interface DBListener. Dans le cas d'un cluster, il est nécessaire de re-déclencher l'appel à ces DBListener sur chaque réplica. Pour cela, JCMS dispose d'un système de type producteur/consommateur capable de diffuser les événements de base de données. Lorsqu'un DBListener est déclenché sur un réplica, celui-ci le sérialise et l'enregistre dans la table DBEventLog. Tous les réplicas surveillent cette table. Lorsqu'un nouvel événement arrive, il déclenche l'appel aux DBListener.

1.3.4 Limites de données stockées dans JcmsDB

Les données stockées dans JcmsDB ont certaines limites par rapport aux données stockées dans JStore :

- Pas d'accès aux données supprimées
- Pas d'extraInfo
- Accès à l'historique de données uniquement pour les classes implémentant HistorizedDBData

Certaines limites sont spécifiques aux classes dérivant de la classe Publication :

- Pas de copie de travail
- Pas de champ multilingue
- Pas de droits de modification
- Pas de gestion des liens inverses
- Pas d'import incrémental
- Pas de choix du gabarit d'affichage
- Les champs textes (textarea) sont par défaut limités à 64 Ko
- Pas d'URL intuitive
- Pas de QueryFilter

Certaines Portlet ont des limites sur le traitement des données de JcmsDB :

- Portlet Requête / Itération (et portlets dérivées)
  - Certains affinements ne sont pas possible si la requête porte sur JcmsDB
  - Pas de « Premières publication »
  - Pas de possibilité de « sauter » les n premières publications

Enfin, certains modules de JCMS ajoutent des fonctionnalités qui ne sont pas opérationnelles sur les données stockées dans JcmsDB. C'est notamment le cas du module Category Right. Vérifiez la compatibilité dans la documentation du module.

1.4 Répartition des données entre JStore et JcmsDB

La répartition des données entre JStore et JcmsDB se fait techniquement classe par classe. Toutes les instances d'une même classe sont stockées dans l'un ou l'autre des systèmes de stockage.

Dans JCMS, les données sont réparties selon les capacités de chacun des systèmes de stockage.

JStore :

- Données de structures accédées très fréquemment (groupes, catégorie, espaces de travail, membres, portlet, …)
- Publication multilingues (article, FAQ, glossaire, document, …)

JcmsDB :

- Données volumineuses (alertes, suivi des lecteurs, suivi des mises à jour, suivi d'activité, membres, documents, tâches, notes de worfklow, archives, contenu conversationnel, …)
- Données mises à jour très fréquemment (suivi des connexions, suivi des lecteurs, bureau personnalisé, …)

JCMS livre en standard deux types de données fonctionnellement proches mais avec un mode de stockage différents :

- Les membres : classe Member (JStore) et classe DBMember (JcmsDB)
- Les documents : classes FileDocument (JStore) et la classe DBFileDocument (JcmsDB)

Dans ces deux cas, il est préférable de privilégier le stockage dans JStore sauf si la volumétrie impose de prendre JcmsDB.

1.5 Identifiants des données JCMS

1.5.1 Structure des identifiants

JCMS utilise un système d'identifiant unique pour identifier toute donnée, qu'elle soit stockée dans JStore (avec ou sans réplication) ou dans JcmsDB.

La structure des identifiants des données stockées dans JStore est la suivante :

```
urid + "_" + logicalTime
```

Avec :

- `urid` (*Unique Replica Identifier*) : une chaine de caractère alphanumérique, commençant par une lettre représentant le réplica JSync sur lequel la donnée a été créée. Au-delà de l'utilisation JSync, ce système est aussi utilisé pour distinguer les différentes instances de la webapp sur lesquelles peuvent être produites des données stockées dans JStore : environnement de développement, de recette, de pré-production.
- `logicalTime` : un compteur qui s'incrémente à chaque insertion de nouvelle donnée et lors des synchronisations JSync.

Exemples : `j_1`, `od_12345`, `prod_123456`, `node1_1234567`, …

La structure des identifiants des données stockées dans JcmsDB est la suivante :

```
rowId + "_" + classShortName
```

Avec :

- `rowId` : le numéro d'enregistrement de la donnée dans la table de la base de données. La gestion de numéro est prise en charge par le SGBDR. Toute table de données JCMS possède cette colonne.
- `classShortName` : le nom court de la classe de la donnée.

Exemples : `1_DBFileDocument`, `123_DBMember`, …

En testant le premier caractère de l'identifiant, JCMS peut immédiatement déterminer s'il s'agit d'une donnée stockée dans JStore (caractère alphabétique) ou d'une donnée stockée dans JcmsDB (chiffre).

Grâce à cette structure d'identifiant, JCMS assure leur unicité dans le temps et dans l'espace des zones de stockage.

1.5.2 Références entre objets

Dans le fichier store.xml, toutes les références sur des objets (stockés dans JStore ou JcmsDB) sont représentées par les identifiants.

Dans les tables de JcmsDB, sauf cas particulier, les colonnes contenant des identifiants sur des données JStore ou JcmsDB sont matérialisées par leur identifiant JCMS.

Certains types de données stockés uniquement dans JcmsDB peuvent utiliser des identifiants basés sur les numéros d'enregistrement (on parle alors de clé étrangère).

1.5.3 Identifiants virtuels

L'utilisation d'identifiant de données en "dur" dans les développements spécifiques ou même dans les types de publications (catégorie par défaut et groupes par défaut) entraîne des problèmes de réutilisation d'une webapp JCMS à une autre.

Pour faire face à ce problème JCMS supporte les identifiants virtuels qui fonctionnent sur le principe des variables. L'identifiant virtuel est une chaîne de caractères commençant par le symbole '$'. Lorsqu'un tel identifiant est fourni aux API de JCMS, celui-ci est résolu en recherchant une propriété portant ce nom.

Exemple d'utilisation :

Dans le fichier WEB-INF/data/custom.prop on déclare l'identifiant virtuel :

```
$id.FicheDocumentaire.Nomenclature : c_1234
```

Ensuite on peut utiliser cet identifiant virtuel pour obtenir l'objet associé :

```
Category cat = channel.getCategory("$id.FicheDocumentaire.Nomenclature");
```

1.5.4 Identifiants externes

JCMS peut gérer des données représentant des objets de systèmes externes. Pour cela, on définit une structure d'identifiants externe et on met en place un système de résolution pour ces identifiants. Pour plus de détails, consultez la section 3.1.4.

1.6 Les autres systèmes de stockage

En complément de JStore et JcmsDB, JCMS utilise le système de fichier pour stocker pour certains types de données.

1.6.1 Fichiers déposés

Tous les fichiers déposés dans JCMS sont stockés sur le système de fichier dans le répertoire upload. Ce répertoire est organisé en sous répertoires. La gestion de cette organisation est entièrement prise en charge par JCMS (cf. section 7)

1.6.2 Propriétés

JCMS utilise plusieurs fichiers de propriétés. Ces fichiers sont principalement utilisés pour :

Les libellés des interfaces localisés dans les différentes langues supportées par JCMS. Ces fichiers de propriétés sont de la forme `<code-de-langue>.prop` (p. ex. `en.prop`, `fr.prop`, `es.prop`, …)

Le paramétrage de certaines fonctionnalités (cœur ou module)

Les principaux fichiers de paramétrage sont :

- `WEB-INF/data/custom.prop` : contient les propriétés éditables depuis les différentes interfaces de paramétrage
- `WEB-INF/jaios/jcms.prop` : contient les valeurs par défaut des propriétés de paramétrage de JCMS
- `WEB-INF/plugins/NOM_DU_MODULE/properties/plugin.prop` : contient les valeurs par défaut des propriétés de paramétrage du module.
- `WEB-INF/data/webapp.prop` : ce fichier n’existe plus depuis JCMS 8. Dans les anciennes versions de JCMS, il contenait des propriétés qui n’étaient pas éditables. Depuis JCMS 8, il est recommandé de mettre ces propriétés dans le module principal du site.

1.6.3 Type

La structure des types de publication est enregistrée dans des fichiers localisés dans le répertoire `WEB-INF/data/types/`. Ce répertoire contient un sous-répertoire par type de publication (ex. PortletNavigation). Chaque sous répertoire contient un fichier XML représentant la structure du type (ex. PortletNavigation.xml) et un fichier XML contenant la liste des gabarits associés à ce type (ex. PortletNavigation-template.xml)

1.6.4 Workflow

La structure de workflow est enregistrée dans des fichiers XML localisés dans le répertoire `WEB-INF/data/workflows/`.

1.6.5 Indexation Lucene

Certaines données bénéficient d’une indexation plein texte. C’est notamment le cas des publications, des catégories, des membres, des fichiers déposés, …

L’indexation est prise en charge par le moteur de recherche Lucene dont les index sont stockés dans le répertoire `WEB-INF/data/lucene/`.

1.6.6 Annuaire LDAP

JCMS peut être connecté avec un annuaire LDAP. Cet annuaire est utilisé pour valider l'authentification des membres, créé des comptes JCMS et mettre à jour les informations. JCMS peut aussi créer des groupes et associés les membres à ces groupes en fonction des rattachements du compte dans l'annuaire LDAP.

JCMS ne fait donc que des lectures de l'annuaire LDAP. Ces lectures sont faites en début de session utilisateur et lorsqu'un administrateur lance une demande de synchronisation LDAP.

1.7 Encodage des caractères

JCMS utilise l'encodage UTF-8 pour toutes les données consommées ou produites :

Toutes les données gérées par JCMS (store, base de données, types de publication, workflow, fichiers de propriétés, ...) sont enregistrées en UTF-8.

Tous les affichages produits par JCMS (page web, mail envoyé, ...) sont encodés en UTF-8 (sauf exception ci-dessous).

La base de données utilisée par JCMS doit être configurée pour utiliser UTF-8.

Exception :

- Export CSV : l'encodage de l'export CSV est par défaut en ISO-8859-1 pour être compatible avec Microsoft Excel. Cet encodage peut être modifié via la propriété csv.charset

1.8 Les codes de langues

L code de langue utilisé dans JCMS est au format ISO-639 custom Jalios BCP-47 : c'est à dire que c'est ISO-639 dans tous les cas, mais pour certaines langues, une sous précision de la norme BCP-47 permet d'y inclure le script.

Exemples :

- Anglais : en (ISO-639)
- Espagnol : es (ISO-639)
- Portugais : pt (ISO-639)
- Chinois (simplifié) : zh (ISO-639)
- Chinois (traditionnel) : zh-Hant (BCP-47)

La liste exhaustive des codes de langue supportés par JCMS se trouve dans jcms.prop, il s'agit des propriétés préfixées par lang.

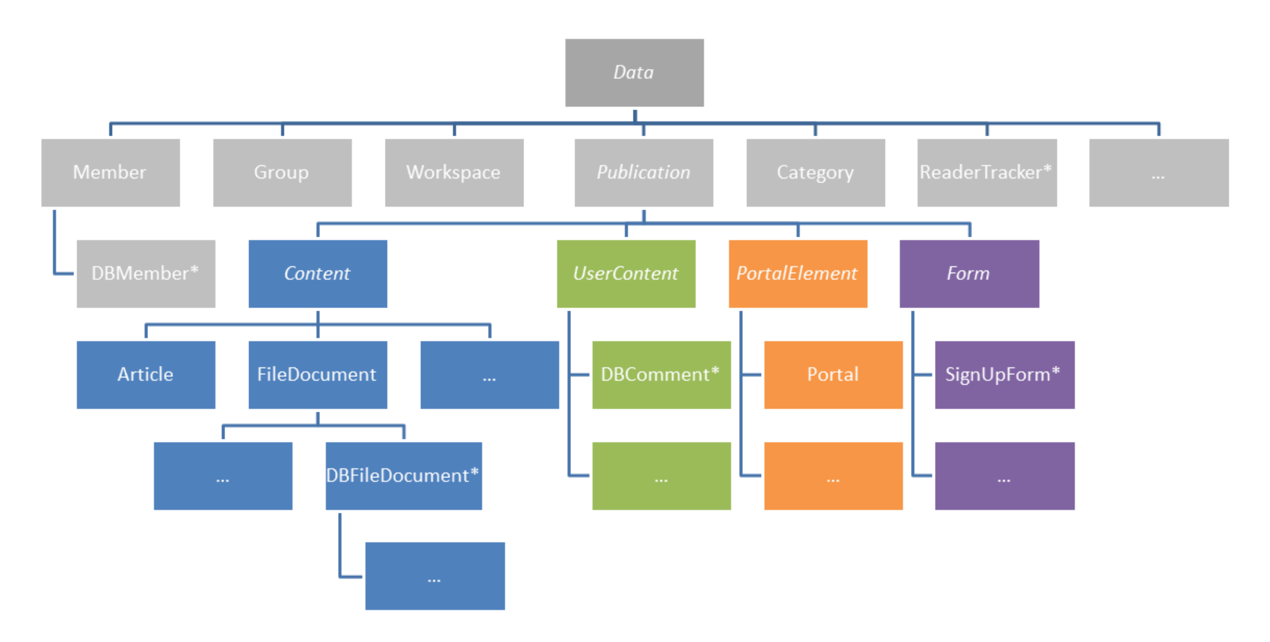
Attention à la casse : Les valeurs des codes de langues sont sensibles à la casse et cela doit être respecté précisément.

2 Le modèle de données de JCMS

2.1 Diagramme des classes

Le diagramme ci-dessous représente les principales classes de données de JCMS et leur relation d'héritage.

Ce diagramme n'est pas exhaustif. Il s'enrichit selon les types de publication que vous avez créés, selon les modules que vous avez installés et selon vos propres développements.



* classe stockée dans JcmsDB

2.2 La classe Data

La classe Data est la classe mère de toutes les données de JCMS. En tant que super classe elle fournit les attributs communs à toutes les données JCMS mais aussi les méthodes. Chaque sous-classe complète ces attributs et ces méthodes selon leur rôle. La classe Data comportent de nombreuses méthodes abstraites qui sont surchargées par ses sous-classes.

2.2.1 Les principaux attributs

Attribut	Description

id	Identifiant unique de cette donnée
cdate	Date de création
mdate	Date de modification
author	Membre auteur de cette donnée
rowId	Numéro d'enregistrement (pour les données de JcmsDB)

2.2.2 Les principales méthodes

Méthodes de représentation de la donnée

Méthode	**Description**
toFullString()	Renvoie une représentation textuelle de la donnée
toCSV()	Renvoie une représentation CSV de la donnée
exportXml()	Renvoie une représentation XML de la donnée
getDataName(language)	Renvoie le nom de la donnée (p. ex. le titre d'une publication, le nom d'un groupe, d'une catégorie, …)
getDataIcon()	Renvoie l'icône associé à la donnée
getDataImage()	Renvoie l'image associée à la donnée
getDisplayLink()	Renvoie le lien HTML associé à la donnée

Méthodes de contrôle sur la donnée

Méthode	**Description**
isPersisted()	Renvoie true si la donnée est enregistrée
checkIntegrity()	Vérifie si la donnée respecte toutes ses contraintes d'intégrité
checkCreate(Member mbr)	Vérifie si la création de la donnée est possible pour le membre mbr.
checkUpdate(Member mbr)	Vérifie si la mise à jour de la donnée est possible pour le membre mbr.
checkDelete(Member mbr)	Vérifie si le membre mbr est autorisé à supprimer la donnée.

Méthodes d'écriture sur la donnée

Méthode	**Description**
performCreate(Member mbr)	Effectue la création de la donnée au nom du membre mbr.
performUpdate(Member mbr)	Effectue la mise à jour de la donnée au nom du membre mbr.
performDelete(Member mbr)	Effectue la suppression de la donnée au nom du membre mbr.
getUpdatInstance()	Renvoie l'objet à utiliser pour la mise à jour de cette donnée

2.2.3 Tag associés

Tag	**Description**
<jalios:checkIntegrity>	Affiche un icône lorsque les contraintes d'intégrité ne sont pas respectées sur la donnée fournie.
<jalios:dataicon>	Affiche l'icône associé à la donnée fournie.
<jalios:edit>	Affiche une icône d'édition pour la donnée fournie.
<jalios:link>	Affiche un lien vers la donnée fournie.
<jalios:lock>	Affiche une icône lorsque la donnée est verrouillée.

2.2.4 ExtraData et ExtraDBData

Les ExtraData et ExtraDBData permettent d'ajouter des champs aux types de données existants sous forme d'une association clé/valeur.

Retenez cette solution pour des enrichissements simples des types natifs de JCMS (Member, Category, Group, …).

Les valeurs sont de type String. C'est donc au développeur de prendre en charge la représentation d'autres types.

Les ExtraData sont représentées par une Map stockée dans les attributs de l'objet. Elles ne sont utilisables que sur les types persistés dans JStore. Chaque modification de cette Map déclenche une opération de type Update dans JStore. Réservez en donc l'usage pour les cas où les données peu volumineuses et avec peu de mises à jour.

Les ExtraDBData sont stockées dans JcmsDB. Il s'agit d'une table associant une valeur à une donnée stockée dans JStore ou JcmsDB. Les ExtraDBData sont donc destinées à enrichir des données de JcmsDB mais aussi de JStore, dans le cas où il s'agit informations volumineuses ou fréquemment mises à jour.

Pour plus de détails sur les ExtraData, consultez l'article *Développer avec les ExtraData et les ExtraDBData* (http://community.jalios.com/howto/extradata).

2.2.5 ExtraInfo

Toute Data comporte une map nommée extraInfo qui permet de stocker de l'information additionnelle sur cette donnée. Cette map n'est pas persistée. Elle ne sert donc qu'à stocker une information calculée et transitoire.

Il y a deux limites d'usage sur les ExtraInfo :

1. Ne les utiliser que sur les objets du Store car les objets de JcmsDB sont gérés par Hibernate. Les objets représentant les données ne sont donc pas gardés en mémoire d'une requête à l'autre (ils peuvent l'être avec l'emploi du cache de second niveau mais il n'y a aucune garantie).
2. Ne pas y stocker directement d'autres données JCMS mais seulement leur pointeur. Ceci afin de ne pas empêcher le *garbage collector* de faire son travail de nettoyage.

2.3 La classe Group

La classe Group représente un groupe de membres.

Les groupes ont plusieurs caractéristiques cumulables :

- Groupe d'espace : Un groupe peut être attaché à un espace de travail (Workspace) ou transverse à tous les espaces. Un membre appartient à un espace de travail s'il appartient à au moins un groupe de cet espace.
- Groupe LDAP : un groupe LDAP est un groupe qui est synchronisé avec un groupe de l'annuaire LDAP. Pour plus de détails, consultez l'article *Configuration et fonctionnement du LDAP dans JCMS* (http://community.jalios.com/ldap).

- Groupe hiérarchique : Un groupe peut aussi avoir un ou plusieurs groupes parents. Dans ce cas, les membres attachés à ce groupe seront systématiquement attachés aux groupes parents et, de proche en proche, à l'ensemble des groupes ancêtres. Les membres bénéficient ainsi de toutes les caractéristiques attachées à ces groupes, notamment les droits.

Avec les membres, les groupes sont au cœur du système de contrôle des droits de JCMS :

- Ils définissent des droits de contributions sur les différents types de publication
- Ils définissent les droits d'usage et de gestion des catégories
- Ils peuvent être associés à une ACL (Access Control List), donnant à leurs membres les droits correspondants à l'ACL
- Ils sont utilisés dans plusieurs fonctions de JCMS et des modules pour déterminer des droits (p. ex. droits de consulter/modifier une publication ou une catégorie, rôle dans les workflows, gestion des invités dans les espaces collaboratifs, …)
- Enfin, les groupes ont eux-mêmes un certain niveau de visibilité : visible de tous, uniquement des membres qui les composent ou uniquement de leurs administrateurs.

2.3.1 Les principaux attributs

Attribut	Description
workspace	Espace de travail auquel est rattaché le groupe.
parentSet	Parents du groupe.
name	Nom du groupe.
ldapDN	Identifiant (DN) LDAP
order	Numéro d'ordre du groupe (pour le tri)
cookieMaxAge	Durée maximum du cookie d'authentification pour les membres de ce groupe.

2.3.2 Les principales méthodes

Méthode	Description
getName(language)	Renvoie le nom du groupe
getChildrenSet()	Renvoie les groupes fils du groupe
getDescendantSet()	Renvoie les groupes descendants du groupe
isAncestor(Group)	Détermine si le groupe est un ancêtre d'un groupe donné
isDescendant(Group)	Détermine si le groupe est un descendant d'un groupe donné
isSubGroup()	Renvoie true si ce groupe a au moins un parent
isLdapGroup()	Renvoie true si c'est un groupe LDAP
getMemberSet()	Renvoie l'ensemble de Membre (JStore) appartenant à ce groupe.
getDBMemberCount()	Renvoie le nombre de DBMember appartenant à ce groupe et ses descendants.
getPublicationSet()	Renvoie l'ensemble des publications attachées aux membres appartenant à ce groupe.
getVisibility()	Renvoie le niveau de visibilité du groupe.
canBeReadBy(Member mbr)	Détermine si le membre mbr peut voir ce groupe.

2.4 La classe Member

La classe Member représente une personne connue de JCMS. Les instances sont stockées dans JStore. La classe dérivée DBMember représente un membre stocké dans JcmsDB.

Les membres ont plusieurs caractéristiques exclusives :

- Compte utilisateur: la classe Member sert essentiellement à gérer des utilisateurs authentifiés. On parle alors de compte utilisateur. Lorsque le compte est actif, il est possible de l'utiliser pour s'authentifier sur le site. Un membre actif peut être synchronisé avec un utilisateur LDAP. L'association est basée sur le login du membre. Pour plus de détails, consultez l'article *Configuration et fonctionnement du LDAP dans JCMS* (http://community.jalios.com/ldap).
- Compte utilisateur désactivé : compte utilisateur avec lequel il n'est plus possible de s'authentifier sur le site.
- Contact : la classe Member sert aussi à représenter une personne externe qui n'a pas d'existence active sur le site. On parle alors de contact. Un contact n'a pas d'identifiant de compte (login) ni de mot de passe. Il ne peut pas s'authentifier sur le site. Cette notion de contact est essentiellement utilisée par le module ESN.
- Invités : le module Espace Collaboratif ajoute un nouveau type de compte utilisateur : les invités. Les invités sont des membres à part entière mais ils ne peuvent naviguer que dans les espaces collaboratifs auxquels ils appartiennent. Pour plus de détails, consulté la page de ce module (http://community.jalios.com/plugin/collaborativespace).

2.4.1 Les principaux attributs

Attribut	Description
name	Le nom de famille.
firstName	Le prénom.
login	Identifiant du compte utilisateur.
email	L'e-mail.
phone	Le numéro de téléphone.
mobile	Le numéro de téléphone portable.
language	La langue du membre (code ISO-639).
street	La rue de l'adresse
postalCode	Le code postal de l'adresse
poBox	La boite postale de l'adresse
region	L'état ou la province de l'adresse
locality	La ville de l'adresse
country	Pays d'appartenance du membre.
organisation	L'organisation ou l'entreprise du membre.
department	Le département.
jobTitle	Le titre de son poste.
password	Le mot de passe (crypté).
photo	Le chemin du fichier contenant la photo.
usage	Indique s'il s'agit d'un compte utilisateur ou d'une fiche contact.
declaredGroups	La liste des groupes auxquels est directement rattaché le membre.

2.4.2 Les principales méthodes

Méthode	**Description**
belongsToGroup (Group grp)	Renvoie true si le membre appartient à un groupe donné.
belongsToWorkspace(Workspace ws)	Renvoie true si le membre appartient à un espace de travail donné.
canPublish(Class c)	Renvoie true si le membre peut publier des instances de la classe donnée. Voir la section sur la gestion des droits d'accès.
canRead(Publication pub)	Renvoie true si le membre peut consulter la publication en paramètre. Voir la section sur la gestion des droits d'accès.
canUseCategory(Category cat)	Renvoie true si le membre peut utiliser la catégorie en paramètre. Voir la section sur la gestion des droits d'accès.
canWorkOn(Publication pub)	Renvoie true si le membre peut modifier ou supprimer la publication en paramètre. Voir la section sur la gestion des droits d'accès.
getAlertList()	Renvoie la liste des alertes qu'a reçues le membre.
getFullName()	Renvoie le nom complet du membre (prénom nom).
getFriendlyName()	Renvoie le nom à utiliser pour s'adresser à ce membre (généralement son prénom).
getGroups()	Renvoie l'ensemble des groupes auxquels appartient le membre.
getLocale()	Renvoie la Locale du membre.
getLastSyncDate()	Renvoie la date de la dernière synchronisation LDAP.
getWorkspaceSet()	Renvoie l'ensemble des espaces de travail auxquels a accès ce membre.
hasPhoto()	Renvoie true si le membre a une photo.
isAccount()	Renvoie true si ce membre représente un compte utilisateur.
isAdmin()	Renvoie true si ce membre est administrateur du site.
isAdmin(Workspace ws)	Renvoie true si ce membre est administrateur d'un espace de travail donné.
isContact()	Renvoie true si ce membre représente une fiche contact.
isDisabled()	Renvoie true si ce membre a été désactivé.
isLdapAccount()	Renvoie true si ce membre est synchronisé avec LDAP.
isValidAccount()	Renvoie true si ce membre est un compte valide qui peut s'authentifier sur le site.
isWorkAdmin()	Renvoie true si ce membre est administrateur d'au moins un espace de travail.
isWorker()	Renvoie true si ce membre est contributeur dans au moins un espace de travail.

2.4.3 Tag associés

Tag	Description
<jalios:login>	Affiche l'interface pour se connecter et se déconnecter.
<jalios:memberphoto>	Affiche la photo du membre.

2.5 La classe DBMember

La classe DBMember dérive de la class Member. Elle est utilisée pour gérer des personnes dont les informations sont stockées dans JcmsDB. Elle permet ainsi de gérer des volumétries d'utilisateurs beaucoup plus importantes que dans le store (plusieurs centaines de milliers).

Du fait de l'héritage, les DBMember ont les mêmes attributs et méthodes que les Member. Cependant, ils ont certaines restrictions (en plus des limitations sur les DBData) :

- Pas de synchronisation LDAP
- Pas la possibilité d'être administrateur central ou administrateur d'un espace de travail
- Pas de délégation d'authentification
- Pas de gestion de droits avec l'audiencement ou le module CategoryRight
- Pas d'affectation aux rôles ouverts des Workflows
- Performances moins élevées avec les Member (dues aux requêtes SQL nécessaires à leur traitement)
- A partir de JCMS 9, les DBMember ne sont plus limités à une quinzaine de groupes.

2.6 La classe Workspace

La classe Workspace représente un espace de travail.

Un espace de travail est composé d'un ou plusieurs groupes de membres. C'est par l'appartenance à un de ces groupes que l'on détermine si un membre appartient à l'espace.

Un ou plusieurs membres sont déclarés administrateurs de l'espace. Ce statut leur donne accès à toutes les données sur leur espace.

Au sein d'un espace, on définit les types de publication utilisés pour la contribution ainsi que leur paramétrage (workflow, droits par défaut, catégorie par défaut, …) Il est possible de clore la contribution dans un espace.

Dans les sites multilingues, une langue par défaut peut être définie sur l'espace. Si c'est le cas, les nouvelles publications seront par défaut créées dans cette langue.

Un quota disque peut être défini pour chaque espace. Au-delà de ce quota, le dépôt de document sera bloqué.

Un espace peut servir de modèle pour créer de nouveaux espaces.

Les espaces peuvent être composés hiérarchiquement. Un espace peut comporter un ou plusieurs sous-espaces qui eux même peuvent comporter des sous-espaces. Un sous-espace ne peut être associé qu'à un seul espace parent. Cette organisation est notamment utilisée par le module Espaces Collaboratifs qui permet au sein d'une communauté d'ouvrir des sous-communautés ou des espaces projets. Les sous-espaces ont une grande autonomie vis-à-vis de leurs espaces parents en ce qui concerne les administrateurs, les participants, les politiques de droits, … Enfin, la recherche de publications faite au sein d'un espace peut être étendue à ses sous-espaces.

2.6.1 Les principaux attributs

Attribut	Description
administrators	Les administrateurs de l'espace.
catSet	L'ensemble des catégories racines de cet espace.
defaultGroup	Le groupe par défaut de l'espace.
language	La langue principale de l'espace.
parent	L'espace parent de l'espace.
tagRoot	La racine des tags de l'espace.
title	Le titre de l'espace.
typeMap	Paramétrage des types utilisés dans l'espace.
Quota	Quota de documents autorisés dans l'espace.
isClosed	Indique si l'espace est fermé.
isModel	Indique si l'espace sert de modèle.

2.6.2 Les principales méthodes

Méthode	Description
getHomeUrl()	Renvoie l'URL de la page d'accueil de cet espace.
getMemberCount()	Renvoie le nombre de membres qui appartiennent à cet espace.
getMemberSet(includeDBMember)	Renvoie l'ensemble des membres qui appartiennent à cet espace. Si includeDBMember est à true, les DBMember appartenant à cet espace sont aussi retournés.
getPublicationSet(Class)	Renvoie l'ensemble des publications de cet espace qui sont des instances de la classe fournie.

2.7 La classe Category

La classe Category représente une catégorie. Les catégories servent au classement des publications. Une publication peut être attachée à une ou plusieurs catégories.

Les catégories sont très utilisées dans le fonctionnement du portail de JCMS. C'est généralement une catégorie qui est pointée lors d'une demande d'affichage d'un portail. A partir de cette catégorie, JCMS détermine le portail à présenter.

Les catégories sont composées hiérarchiquement. Cette hiérarchie est utilisée dans les recherches : on peut rechercher les publications attachées à la catégorie ou à l'une de ses descendantes.

Plusieurs droits opèrent sur les catégories :

- Droits d'utilisation : seuls les membres autorisés peuvent classer des publications dans cette catégorie et sa descendance.
- Droits de consultation : seuls les membres autorisés peuvent voir cette catégorie et sa descendance.
- Droits de gestion : seuls les membres autorisés peuvent voir gérer cette catégorie et sa descendance (ajouter, modifier, déplacer et supprimer des catégories).

2.7.1 Les principaux attributs

Attribut	Description
name	Le nom de la catégorie.

description	La description de la catégorie.
parent	La catégorie parent à laquelle est rattachée la catégorie
icon	L'icône de la catégorie
image	L'image de la catégorie.
order	Le numéro d'ordre de la catégorie par rapport à ses catégories sœurs.
synonyms	Les synonymes de la catégorie
isExclusive	Indique que cette branche de catégorie contient des catégories qui doivent être sélectionnées de façon exclusive pour une publication.
isSelectable	Indique que des publications peuvent être rattachées directement à cette catégorie
authorizedMemberSet	Ensemble des membres (JStore) autorisés à voir cette catégorie.
authorizedGroupSet	Ensemble des groupes autorisés à voir cette catégorie.

2.7.2 Les principales méthodes

Méthode	**Description**
isNode()	Renvoie true si la catégorie contient des catégories filles.
isLeaf()	Renvoie true si la catégorie ne contient pas de catégories filles.
getChildrenSet()	Renvoie la liste des catégories filles.
getDescendantSet()	Renvoie la liste de toutes les catégories descendantes de cette catégorie.
hasAncestor(Category cat)	Renvoie true si cette catégorie a la catégorie fournie parmi ses ancêtres.
containsDescendant(Category cat)	Renvoie true si la catégorie est ancêtre de la catégorie fournie.
isSibling(Category cat)	Renvoie true si la catégorie et celle fournie sont sœurs.
getContentSet()	Renvoie l'ensemble des contenus attachés directement à cette catégorie.
getAllContentSet()	Renvoie l'ensemble des catégories attachées à cette catégorie ou à l'une de ses descendantes.

2.8 La classe Publication

La classe Publication représente une publication. Cette classe abstraite regroupe l'ensemble des attributs et des méthodes communs aux classes qui en dérivent. On trouve parmi elles :

- les classes de contenu : Article, Brève, page Wiki, Document, …
- les classes de contenus utilisateurs : commentaires, information de profil utilisateur, …
- les classes des formulaires : demande d'inscription, demande d'ouverture d'espace collaboratif, …
- les classes des portlets

Certaines classes dérivées de la classe Publication sont stockées dans JStore et d'autres dans JcmsDB.

Les principales caractéristiques d'une publication :

- Elle a un titre
- Elle a des catégories
- Elle est rattachée à un espace de travail

- Elle a des droits de consultation et de mise à jour
- Elle a une date de mise à jour à majeure, une date de publication, une date d'expiration et une date d'archivage
- Elle est rattachée à un workflow
- Elle est dans un certain état du workflow (représenté par l'attribut pstatus)
- On peut connaître les membres qui ont consulté la publication
- On peut connaître les membres qui suivent les modifications de la publication
- Elle peut disposer de différents gabarits d'affichage
- Elle regroupe des sous classes de Comparator et de DataSelector
- Le contenu de la publication est indexé dans Lucene
- Toute publication implémente la classe TreeNode qui représente une arborescence. Certains types de publications exploitent cette arborescence.

2.8.1 Les principaux attributs

Attribut	**Description**
title, titleML	Titre.
categories	Catégories.
workspace	Espace de travail.
pstatus	Etat de workflow.
pdate	date de publication.
edate	date d'expiration.
adate	date d'archivage.
udate	date de mise à jour majeure.
sdate	date de tri.
authorizedMemberSet, authorizedGroupSet	Droits de consultation.
updateMemberSet, updateGroupSet	Droits de mise à jour.
mainLanguage	Langue principale de la publication.
mainInstance	Instance d'origine des copies de travail.

2.8.2 Les principales méthodes

Méthode	**Description**
canBeReadBy(Member)	Renvoie true si le membre peut consulter la publication.
hasBeenReadBy(Member)	Renvoie true le membre a consulté la publication.
getWeakReferrerSet()	Renvoie l'ensemble des publications qui référence cette publication via une référence « molle » (URL, lien wiki, …)
getWorkflow()	Renvoie le workflow de cette publication.
isInVisibleState()	Renvoie true si la publication est dans un état visible.
isTracked()	Retourne true si les lecteurs qui consultent la publication sont suivis.
trackReader(Member)	Indique que le membre a consulté la publication.
getAbstract()	Renvoie le champ résumé de la publication.
getTreeChildren()	Renvoie les publications filles de cette publication.
getTreeParent()	Renvoie la publication parente de cette publication.
isTreeLeaf()	Renvoie true si cette publication n'a pas de publications filles.
isTreeNode()	Renvoie true si cette publication a des publications filles.
getTreeRoot()	Renvoie la racine de publication de cette publication.
getDataImage()	Renvoie la première image trouvée dans les champs wiki et wysiwyg de la publication. Cette implémentation par défaut peut etre surchargée

	par certaines sous classes.

2.9 La classe Content

La classe Content regroupe les publications représentant un contenu éditorial. Il s'agit d'une classe de typage. Elle n'ajoute aucun nouveau attribut ni aucune méthode.

Pour bénéficier au maximum des possibilités de JCMS (droits, multilinguisme, …) les contenus sont généralement stockés dans JStore.

2.10 La classe UserContent

La classe UserContent regroupe les publications représentant un contenu produit par un utilisateur sans droit particulier. Il s'agit d'une classe de typage. Elle n'ajoute aucun nouvel attribut ni aucune méthode.

En standard, les contenus utilisateurs ne nécessitent pas de droit de contribution et n'ont pas de droits de consultation.

Du fait de leur volumétrie potentielle, les contenus utilisateurs sont plutôt stockés dans JcmsDB mais ce n'est pas une obligation.

2.11 La classe Form

La classe Form représente la soumission d'un formulaire. Les soumissions pouvant être anonymes, l'auteur de l'objet soumission n'est pas la personne qui a effectué la soumission mais le responsable des soumissions.

2.11.1 Les principaux attributs

Attribut	Description
submitMember	Le membre qui a soumis le formulaire
submitRemoteAddr	L'adresse IP du soumissionnaire

2.12 La classe PortalElement

La classe PortalElement est la classe mère de toutes les portlets.

2.12.1 Les principaux attributs

Attribut	Description
abilities	Aptitudes de la portlet
cacheTypeInt	Type de cache de la portlet
behaviorCopy	Mode de copie de la portlet
cssClasses	Classes CSS associées à la portlet
portletImage	Image associée à la portlet

2.13 La classe FileDocument

La classe FileDocument dérive de Publication. Elle représente un document, c'est-à-dire un fichier qui a été déposé sur le site accompagné d'un ensemble de métadonnées.

Les principales caractéristiques d'un document :

- Il référence un fichier sur disque.
- Il a une date de (dernier) dépôt.
- Le contenu du fichier peut être indexé.
- Lorsqu'il est référencé par une autre publication, son état de workflow et ses droits de consultation sont assujettis à cette publication.
- Un PDF peut être associé au document. Ce PDF peut être soit déposé manuellement soit généré par le module Convertisseur PDF.
- A partir de JCMS 9, les types générés peuvent dériver de FileDocument.

A partir de JCMS 9, un FileDocument peut être associé à n'importe quel workflow. S'il est associé au Workflow des pièces jointes, alors le document est considéré comme une pièce jointe. Il est alors asservi par les publications qui le référencent. Si toutes les publications qui le référencent sont dans un état non visible du workflow, alors le FileDocument est placé dans l'état Caché du Workflow des pièces jointes. Dès que l'une des publications référentes devient visible, le FileDocument est placé dans l'état Publié. De même à partir de JCMS 9 SP1, les documents qui suivent le Workflow des pièces jointes ont leurs droits d'accès asservis aux publications qui les référencent.

2.13.1 Les principaux attributs

Attribut	Description
description / descriptionML	La description multilingue du document.
filename	Le chemin du fichier, relatif à la webapp.
contentType	Le type mime du fichier.
originalFilename	Le nom du fichier tel qu'il a été reçu lors du dépôt.
uploadDate	Date de dépôt du fichier.
pdfUploadDate	Date de dépôt du PDF associé à ce fichier.

2.13.2 Les principales méthodes

Méthode	Description
getAssociatedFileSet()	Retourne l'ensemble des fichiers associés à ce document.
getAssociatedPDF()	Retourne le PDF associé à ce document.
getDimensions()	Retourne les dimensions de ce document (pour les images et les vidéos).
getDuration()	Retourne la durée de ce document (pour les audio).
getFile()	Retourne le File représentant le fichier du document.
getFileReferrerSet()	Retourne toutes les données JCMS qui référencent ce fichier
getGenericContentType()	Retourne la première partie du content-type (p. ex. « video » si le content-type est « video/mpeg »).
getGenericThumbnail()	Retourne une vignette générique pour ce document.
getHeight()	Retourne la hauteur du document (pour les images).
getIndexedDate()	Retourne la date d'indexation du fichier.
getLastModified()	Retourne la date de dernière modification du fichier.
getLuceneDocument()	Retourne l'objet Lucene associé à ce document

getMediaType()	Retourne « image », « audio », « video », « flash » ou « other » selon le type de document.
getMetaData(String)	Retourne les métadonnées associées au fichier (et extraites via un MetaDataExtractor).
getSize()	Retourne le poids du fichier.
getTypeInfo(lang)	Retourne une description du type fichier localisé dans la langue fournie.
getWidth()	Retourne la largeur du document (pour les images).
isAudio()	Retourne true si le document est un audio.
isFlash()	Retourne true si le document est un flash.
isImage()	Retourne true si le document est une image.
isIndexed()	Retourne true si le contenu du fichier a été indexé.
isMedia()	Retourne true si le document est un média (image, video, audio, flash).
isText()	Retourne true si le document est un texte.
isVideo()	Retourne true si le document est une vidéo.
isWebImage()	Retourne true si le document est une image dans un format affichable dans un navigateur (GIF, JPG, PNG)
supportsThumbnail()	Retourne true si des vignettes peuvent être générées pour ce document.

2.13.3 Les principales méthodes statiques

Méthode	**Description**
getInstance(…)	Retourne un nouveau FileDocument correspondant au fichier fourni. Le FileDocument n'est pas enregistré dans le Store.
getNewDocumentDirectory(contentType)	Retourne le répertoire où doit être classé le fichier du type de contenu indiqué.

2.13.4 Extraction des métadonnées

Si le fichier comporte des métadonnées (p. ex. les exif pour les photos), elles peuvent être extraites via un MetaDataExtractor.

Par défaut, JCMS ne fournit que le MetaDataExtractor pour les images (pour obtenir les dimensions et les exif).

Pour extraire les métadonnées d'autres types de fichier, il faut développer une classe implémentant l'interface MetaDataExtractor et la déclarer par une propriété de la forme file-document.mde.<content-type>.

Exemple :

```
file-document.mde.video/mpeg : com.example.MyVideoMetaDataExtractor
```

Les méta-données ainsi extraites sont mises en cache et accessibles par la méthode getMetaData().

2.14 La classe DBFileDocument

La classe DBFileDocument dérive de FileDocument mais implémente DBData. Les instances de cette classe sont donc stockées dans JcmsDB. Elle implémente aussi les interfaces suivantes :

- CategorizedDBData (pour avoir le support des catégories)
- HistorizedDBData (pour avoir la gestion de l'historique de version)

- OpenRoleDBData (pour pouvoir définir des rôles ouverts)
- TrackedDBData (pour permettre le suivi des lecteurs)

Le nombre total de DBFileDocument qui peuvent être créés est contrôlé par JCMS et ne peut dépasser la limite définie dans le addPack du site (par défaut limité à 100 DBFileDocument).

2.14.1 Les principales méthodes statiques

Méthode	Description
getInstance(…)	Retourne un nouveau DBFileDocument correspondant au fichier fournie. Le DBFileDocument n'est pas enregistrée dans le JcmsDB.

2.15 Les interfaces de typage

JCMS propose plusieurs interfaces de typage qui confèrent aux classes qui les implémentent un comportement particulier.

2.15.1 L'interface DBData

Toute classe de donnée qui doit persister ses données dans JcmsDB doit implémenter l'interface DBData.

Cette interface permet d'accéder à l'attribut rowId.

2.15.2 L'interface CategorizedDBData

Par défaut, les classes de données dérivant de Publication et persistées dans JcmsDB n'ont pas le support des catégories. Pour l'activer il faut implémenter l'interface CategorizedDBData.

Lorsque c'est le cas, deux tables de collections sont créées pour gérer les catégories de la publication :

- `catIdSet` : cette collection contient les catégories de la publication
- `allCatIdSet` : cette collection contient les catégories de la publication ainsi que toutes les catégories ancêtres de ses catégories. Elle permet de rechercher les publications attachées à une sous-catégorie d'une catégorie donnée.

2.15.3 L'interface HistorizedDBData

Par défaut, les classes de données dérivant de Publication et persistées dans JcmsDB ne sont pas historisées. Pour activer l'historique des versions, la classe doit implémenter l'interface HistorizedDBData.

Lorsque c'est le cas, une classe dédiée à cet usage est créée pour la classe historisée. Elle reprend le nom de cette classe avec le suffixe « Revision ». P. ex. DBFileDocumentRevision. A chaque écriture sur la donnée un enregistrement est ajouté dans DBFileDocumentRevision.

Si JcmsDB est gérée avec le SGBDR Oracle, il faut faire attention à la longueur des noms des classes des DBData car ils servent à générer les noms des tables. Or dans Oracle le nom des tables est limité à 30 caractères. Par ailleurs JCMS préfix toutes les table avec 2 caractères (« j_ »). D'une manière générale, JCMS contrôle la longueur du nom des tables du type générés mais pas celui de la table

des révisions. Aussi, lorsque vous déclarez un type stocké dans JcmsDB avec l'historique des versions activé, choisissez un nom compatible avec cette contrainte.

2.15.4 L'interface TrackedDBData

Par défaut, les classes persistées dans JcmsDB n'ont pas le suivi des lecteurs (ReaderTracker). Seules les classes dérivant de Content en bénéficient automatiquement. Pour l'activer sur les autres types de Publication, il faut implémenter l'interface TrackedDBData.

2.15.5 L'interface EditableData

L'interface EditableData indique que la classe dispose de gabarits d'édition. Seules les classes qui implémentent cette interface peuvent être éditées avec le tag `<jalios:edit/>`.

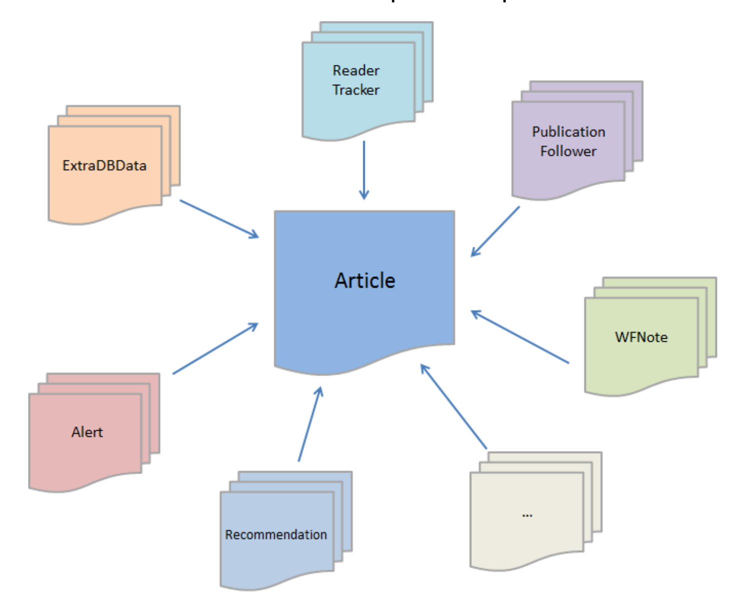
2.15.6 L'interface StrongLockable

L'interface StrongLockable indique que la classe supporte les verrous durs (StrongLock). En standard, seuls les contenus implémentent cette interface.

2.16 Les classes de données satellites

Certaines classes de données sont rattachées à une donnée. Elles ne sont pas agrégées dans la donnée car elles sont généralement stockées dans JcmsDB pour des raisons de volumétrie ou de mises à jour. Elles peuvent porter sur des données stockées dans JStore ou JcmsDB.

C'est par exemple le cas dans l'exemple ci-dessous, où de nombreuses classes gravitent autour d'un Article. Chacune de ces classes pointe la publication au travers de son identifiant.



En plus des classes de JCMS, des modules peuvent ajouter leurs propres classes satellites. C'est par exemple le cas du module Commentaire (un commentaire est attaché à un contenu) ou du module Gestion de tâches (une tâche peut être attachée à une publication).

2.16.1 La classe ReaderTracker

La classe ReaderTracker contient les informations de suivi de lecture sur une publication. Une instance contient les informations des accès d'un membre sur une publication.

2.16.2 La classe PublicationFollower

La classe PublicationFollower contient les informations pour le suivi des changements d'une publication. Une instance contient l'identifiant d'un membre qui suit la publication.

2.16.3 La classe WFNote

La classe WFNote contient les informations associées au changement d'état d'une publication. Elle comporte notamment l'état de départ, l'état d'arrivée, l'auteur du changement d'état et un éventuel commentaire.

2.16.4 La classe Recommendation

La classe Recommendation contient les informations d'une recommandation (donnée recommandée, auteur, destinataires, texte, …)

2.16.5 La classe ExtraDBData

La classe ExtraDBData permet d'ajouter un attribut à une donnée. Elle est composée d'un identifiant de la donnée, d'une clé et d'une valeur. C'est la seule donnée de JCMS qui ne dérive pas de la classe Data (pour réduire la volumétrie de stockage).

2.16.6 La classe Alert

La classe Alert représente une alerte persistante qui porte sur une donnée. Les alertes sont personnalisées. Ainsi, si un utilisateur ou une fonction de JCMS déclenche l'envoi d'alerte à 10 personnes, 10 objets Alert seront enregistrés. Pour éviter, de garder trop d'alertes obsolètes, JCMS gère la suppression automatique des alertes au bout d'un certain temps. Pour plus de détails, consultez l'article *API de gestion des alertes* (http://community.jalios.com/howto/alert).

2.17 Les publications importées

JCMS permet d'importer et d'exporter des publications d'autres sites. Les publications importées possèdent un attribut importMap qui contient des informations liées à l'import. Cette map contient entre autre :

- L'identifiant de la donnée d'origine
- La date du dernier import
- Le lot d'import de la donnée
- La date de modification de la donnée d'origine
- La signature de la donnée d'origine
- La source d'import
- L'URL de la donnée d'origine
- L'identifiant de l'auteur d'origine
- La version de la donnée
- Est-ce qu'un conflit de mise à jour a été détecté lors du dernier import ?

Ces informations sont accessibles via la méthode getImportMap() ou en utilisant les méthodes dédiées (isImported(), getImportId(), getImportMdate(), …)

Pour le détail de la mise en œuvre de l'import/export avec JCMS consultez l'article *Mise en œuvre de l'Import / Export* (http://community.jalios.com/howto/importexport).

2.18 Développement de nouveaux types de données

JCMS fournit en standard de nombreux types de données. Cependant, dans certains cas il y a besoin de gérer de nouveaux types de données. Pour cela, plusieurs approches sont possibles.

2.18.1 Type de publication

JCMS intègre un générateur de type de publication.

Utilisez cette approche, si vous devez créer des types de données qui dérivent de l'une des sous-classes de Publication.

Après avoir décrit le type (champs, propriété, …), JCMS génère toutes les ressources nécessaires à son exploitation. Ces ressources suivent le design pattern MVC (Modèle Vue Contrôleur). Cela inclut notamment la classe Java qui représente le type (le modèle), une classe Java pour l'édition de l'objet (le contrôleur) et des ressources pour la présentation (dont une JSP correspondant à la Vue).

Lors de la déclaration vous pouvez préciser de nombreuses informations sur le type :

- Le mode de stockage (JStore / JcmsDB)
- La super classe (par défaut Content)
- Des interfaces à implémenter
- Des options propres à JcmsDB (génération ou non du HBM, historisation, utilisation des catégories, …)

De même pour chaque champ, il est possible de préciser :

- Le type de champ (texte, date, nombre, lien, …) ce qui va déterminer le type Java du champ, son mode d'édition et de représentation.
- La cardinalité du champ (monovalué / multivalué)
- Des options propres au type du champ

2.18.2 Types spécifiques

Enfin, il est bien entendu possible de créer des nouveaux types de données en codant une nouvelle classe Java. Celle-ci doit dériver de Data ou de l'une de ses sous-classes. Elle doit implémenter l'interface DBData pour être stockée dans JcmsDB (et déclarer un fichier HBM pour le mapping Hibernate).

Utiliser cette approche pour des types de données techniques ou pour des enrichissements sur des types de données existants.

2.19 L'API de gestions de types

Les types générés sont stockés au format XML. JCMS fournit une API pour simplifier l'accès à ces informations ainsi qu'au paramétrage des types dans les espaces de travail.

2.19.1 La classe TypeEntry

La classe TypeEntry représente un type généré. Elle est obtenue par l'appel à la méthode channel.getTypeEntry(Class).

Les principales méthodes

Méthode	Description
getLabel(lang)	Retourne le libellé du type dans la langue lang.
getDescription(lang)	Retourne la description du type dans la langue lang.
getTemplateEntrySet()	Retourne l'ensemble des gabarits disponible pour ce type
isAbstract()	Retourne true s'il s'agit d'un type abstrait.

2.19.2 La classe TypeFieldEntry

La classe TypeFieldEntry représent un champ d'un type généré. Elle est obtenue par l'appel à la méthode channel.getTypeFieldEntry(class, fieldName, checkAll). L'attribut checkAll détermine si il faut aussi rechercher ce champ dans les champs hérités.

Les principales méthodes

Méthode	Description
getEditor()	Retourne le mode d'édition du champ.
getLabel(lang)	Retourne le libellé du champ dans la langue lang.
getName()	Retourne le nom du champ.
getType()	Retourne le type (i.e. la classe) du champ.
isAbstract()	Retourne true s'il s'agit du champ servant de résumé.
isFieldWiki()	Retourne true s'il s'agit d'un champ Wiki.
isFieldWysiwyg()	Retourne true s'il s'agit d'un champ wysiwyg.
isLink()	Retourne true s'il s'agit d'un champ de type lien sur une donnée.

2.19.3 La classe WSTypeEntry

La classe WSTypeEntry représente le paramétrage d'un type dans un espace de travail. Elle est obtenue par l'appel à la méthode workspace.getWSTypeEntry(class).

Les principales méthodes

Méthode	Description
getTemplates()	Retourne les gabarits déclarés pour ce type.
getWorkflow(boolean)	Retourne le workflow utilisé pour ce type.

3 L’accès aux données

Une fois qu’une donnée a été persistée dans JCMS (dans JStore ou dans JcmsDB), il existe de nombreux moyens pour la retrouver.

3.1 Accès à une donnée

3.1.1 Accès par Identifiant

La méthode getData() de la classe Channel permet d’accéder à une donnée à partir de sa classe et de son identifiant. L’identifiant peut être un identifiant JCMS ou un identifiant virtuel.

Exemples :

```
Group grp = channel.getData(Group.class, "j_1");
Member mbr = channel.getData(Member.class, "j_2");
Category cat = channel.getData(Category.class, "j_3");
DBFileDocument doc = channel.getData(DBFileDocument.class, "1_DBFileDocument");
Category cat = channel.getData(Category.class, "$channel.root-category");
```

La classe indiquée en paramètre sert au typage du type de retour. Il est donc possible de mettre une super classe de la donnée.

Exemple :

```
String id = "j_300";
Article article = channel.getData(Article.class, id);
Publication pub = channel.getData(Publication.class, id);
Data data = channel.getData(Data.class, id);
```

La classe Channel fournit aussi des méthodes d’accès dédiées à certains types de données.

Exemples :

```
Group grp = channel.getGroup("j_1");
Member mbr = channel.getMember("j_2");
Category cat = channel.getCategory("j_3");
Workspace ws = channel.getWorkspace("j_4");
Publication pub = channel.getPublication("j_300");
```

Pour les données stockées dans JcmsDB, la méthode getDBData() permet d’y accéder par leur classe et leur rowId.

Exemple :

```
DBMember mbr = channel.getDBData(DBMember.class, 1L);
```

3.1.2 Accès par une autre donnée

Lorsqu'une donnée référence une autre donnée, elle possède généralement une méthode permettant d'accéder à cette donnée.

Par exemple, une publication référence l'espace de travail auquel elle appartient, l'auteur de la publication, les catégories qui y sont attachées, ... Il est donc possible d'accéder à ces données depuis une publication.

Exemples :

```
Publication pub = channel.getPublication("j_300");
Workspace ws = pub.getWorkspace();
Member author = pub.getAuthor();
```

3.1.3 Accès par méthode dédiée

JCMS fournit des méthodes pour accéder à certaines données bien connues du site.

Exemples :

```
Group defaultGroup = channel.getDefaultGroup();
Member defaultAdmin = channel.getDefaultAdmin();
Workspace defaultWS = channel.getDefaultWorkspace();
Category rootCat = channel.getRootCategory();
```

3.1.4 Accès à une donnée externe

JCMS peut avoir à interagir avec des données externes. C'est par exemple le cas avec les modules Exchange, Lotus Notes, Zimbra, CMIS, Google Docs, … Lors d'une recherche ces modules intègrent aux résultats de JCMS les résultats issus du système distant avec lequel ils opèrent.

JCMS peut prendre en charge des données externes comme si il s'agissait de données internes. Pour cela il faut les représenter par des objets dérivant de la classe Publication et leur donner un identifiant. Il ne s'agit ni d'un identifiant JStore ni d'un identifiant JcmsDB mais d'un identifiant externe.

La structure des identifiants externes est libre. Néanmoins, il est recommandé de les préfixer par le service distant qu'il représente, suivi de l'identifiant spécifique de cette plateforme.

Exemple : `CUSTOM_1234567`

Lorsque JCMS essaie de résoudre un identifiant externe il échoue et invoque alors la méthode getData() de la classe ChannelPolicyFilter. Il suffit dans cette méthode d'instancier un objet Publication (p. ex. une WebPage) et de l'alimenter avec les données du système distant. JCMS peut alors le traiter comme si il s'agissait d'une donnée persistée dans JStore ou JcmsDB.

Exemple :

```
public class AmazonChannelPolicyFilter extends BasicChannelPolicyFilter {

  private static final String AMAZON_ID_PREFIX = "AMAZON_";
  private AmazonManager amazonMgr = AmazonManager.getInstance();

  @Override
  public Data getData(String id) {
    if (!id.startsWith(AMAZON_ID_PREFIX)) {
```

```
      return null;
    }
    return amazonMgr.getItemByJcmsId(id);
  }
}
```

3.2 Accès à une collection de données

JCMS fournit de nombreuses méthodes pour obtenir une collection de données à partir d'un certain critère.

3.2.1 Accès par une classe

Plusieurs méthodes sont disponibles pour retrouver toutes les instances d'une classe donnée.

getDataSet()

La méthode getDataSet() renvoie tous les instances de la classe fournie en paramètre. Le TreeSet retourné est trié selon le *natural comparator*. Celui-ci est basé sur la date de création (cdate) puis l'identifiant en cas d'égalité.

Pour les données persistées dans JStore l'accès est quasi instantané car JCMS maintient un index par classe. Pour les données de JcmsDB, une requête est effectuée et les résultats sont déversés dans un TreeSet.

Exemple :

```
Set<Group> allGroupSet = channel.getDataSet(Group.class);
```

Attention ! Le Set retourné **NE DOIT PAS** être modifié. Si vous devez opérer dessus, faite une copie.

Exemple :

```
Set<Group> allGroupSet = channel.getDataSet(Group.class);
Set<Group> copySet = new TreeSet(allGroupSet);
copySet.remove(channel.getGroup("j_1"));
```

getPublicationSet()

La méthode gePublicationSet() renvoie toutes les instances d'une classe de publication et effectue un contrôle de droits. Elle reçoit en paramètre la classe recherchée et un membre. Seules les instances que le membre a le droit de consulter sont retournées. Le TreeSet retourné est trié selon le *natural comparator*.

Du fait du contrôle des droits, cette méthode est bien moins rapide que getDataSet().

Le Set retourné peut être modifié.

Exemple :

```
Member m1 = channel.getMember(id1);
Member m2 = channel.getMember(id2);
Set<Publication> pubSet1 = channel.getPublicationSet(FileDocument.class, m1);
Set<Publication> pubSet2 = channel.getPublicationSet(FileDocument.class, m2);
pubSet1.removeAll(pubSet2); // Keep only publication m1 can access but not m2
```

3.2.2 Accès par index (JStore)

JCMS maintient de nombreux index sur les données JStore. L'accès à ces données est donc très rapide. Les index sont calculés à partir des références des données.

Exemples :

- Une publication référence un membre (l'auteur). JCMS maintient donc un index de toutes les publications d'un membre.
- Une publication référence un espace de travail. JCMS maintient donc un index de toutes les publications d'un espace de travail.
- Une publication référence un ensemble de catégories. JCMS maintient donc un index de toutes les publications associées à une catégorie.

L'index est accessible directement sur l'instance concernée.

Exemple :

```
Group grp = channel.getGroup("j_1");
Set<Member> mbrSet = grp.getMemberSet();

Member mbr = channel.getMember("j_2");
Set<Publication> pubSet = mbr.getPublicationSet();

Workspace ws = channel.getWorkspace("j_4");
Set<Publication> pubSet = ws.getPubSet();
Set<Article> articleSet = ws.getPublicationSet(Article.class);

Category cat = channel.getCategory("j_3");
Set<Publication> pubSet = cat.getPublicationSet();
Set<Publication> pubSet = cat.getAllPublicationSet();
```

Attention ! Le Set retourné NE DOIT PAS être modifié. Si vous devez opérer dessus, faite une copie.

```
Group g1 = channel.getGroup(g1Id);
Group g2 = channel.getGroup(g2Id);
Set<Member> mbrSet = new TreeSet(g1.getMemberSet());
mbrSet.addAll(g2.getMemberSet());
```

Pour les types générés, JCMS maintient un index pour chaque champ de type lien. Grâce à cet index, il est possible de retrouver toutes les instances de publication qui référencent une donnée.

Exemple :

Une Faq est composée de FaqEntry qui représente une question et une réponse. Chaque FaqEntry référence une Faq.

```
Faq faq = channel.getData(Faq.class, "c_1234");
Set<FaqEntry> entrySet = faq.getLinkIndexedDataSet(FaqEntry.class);
```

3.3 Accès par requête

JCMS propose plusieurs solutions pour retrouver une collection de données correspondant à un ou plusieurs critères. Celles-ci varient selon la nature des données recherchées et le type de filtrage à appliquer.

Le filtrage consiste à retenir une sous partie d'une collection de données. Généralement le filtrage s'accompagne d'un tri qui ordonne les résultats trouvés.

3.3.1 Filtrage par DataSelector

Lorsque les données recherchées sont persistées dans JStore, il est possible d'appliquer un filtrage avec la méthode JcmsUtil.select(collection, selector, comparator). Celle-ci reçoit en paramètre une collection de données à filtrer, un DataSelector qui effectue le filtrage et un Comparator qui trie des résultats. La méthode retourne les résultats dans un TreeSet .

Exemple : récupérer l'ensemble des membres désactivés triés par nom.

```
Set<Member> set = channel.getDataSet(Member.class)
DataSelector selector = new Member.DisabledSelector();
Comparator comparator = ComparatorManager.getComparator(Member.class, "name");
Set<Member> resultSet = JcmsUtil.select(set, selector, comparator);
```

Exemple : récupérer l'ensemble des articles publiés depuis 31 jours, triés par date de publication

```
Set<Article> set = channel.getDataSet(Article.class);
Date lastMonth = new Date(System.currentTimeMillis() - MILLIS_IN_ONE_MONTH);
DataSelector selector = Publication.getPdateSelector(lastMonth, null));
Comparator comparator = ComparatorManager.getComparator(Publication.class, "pdate");
Set<Member> resultSet = JcmsUtil.select(set, selector, comparator);
```

Choix de la collection de données

Pour la collection de données, vous pouvez utiliser n'importe quelle collection (Set ou List). Le temps du filtrage étant proportionnel au nombre d'éléments de la collection, travaillez toujours sur la collection la plus précise possible. Par exemple, utilisez les index JStore disponibles.

Choix d'un DataSelector existant

JCMS fournit de nombreux DataSelector prêts à l'emploi. Ils sont généralement sous forme d'inner classes dans les données sur lesquelles ils portent. Dans certains cas, une méthode statique est proposée pour obtenir le DataSelector (p. ex. Publication.getReadRightSelector())

Quelques exemples :

- Category.NodeSelector
- Category.LeafSelector
- Data.AuthorSelector
- Data.DateSelector
- Group.WorkspaceSelector
- Group.ParentSelector
- Member.AccountSelector
- Member.ContactSelector
- Member.DisabledSelector
- Member.EnabledSelector
- Publication.CanWorkOnSelector
- Publication.WorkspaceSelector
- Publication.PdateSelector
- Publication.PstatusSelector
- Publication.ReadRightSelector

- Publication.TrackedSelector
- Publication.VisibleStateSelector
- Workspace.CollaborativeSpaceSelector
- Workspace.OpenWorkspaceSelector
- Workspace.MemberSizeSelector

Ecriture d'un DataSelector

Pour faire des filtrages sur d'autres critères que ceux proposés en standard ou pour des critères plus complexes, il est nécessaire d'écrire son propre DataSelector.

Pour cela, il suffit d'écrire une classe implémentant l'interface DataSelector et son unique méthode isSelected(). Cette méthode reçoit la donnée proposée et retourne true si il faut la retenir et false autrement.

Les DataSelector étant généralement des classes très simples, il est souvent pratique de les déclarer par une classe anonyme.

Exemple : retourner toutes les publications de l'espace WS1 dont les auteurs ne sont pas des comptes actifs.

```
Set<Publication> pubSet = channel.getWorspace(ws1Id).getPubSet();
DataSelector selector = new DataSelector() {
  public boolean isSelected(Data data) {
    Member author = data.getAuthor();
    return author == null || !author. isValidAccount();
  }
}
Set<Publication> resultSet = JcmsUtil.select(pubSet, selector , null);
```

Combinaison des DataSelector

Pour faire une recherche multicritères, il suffit de combiner plusieurs DataSelector.

Pour cela JCMS propose les DataSelector suivants :

- AndDataSelector : vérifie que tous les DataSelector fournis acceptent la donnée.
- OrDataSelector : vérifie qu'au moins un des DataSelector fournis accepte la donnée.
- ReverseDataSelector : retourne le contraire du DataSelector fourni.

Exemple : retourner tous les articles publiés depuis un mois qui sont visibles et qui n'ont pas de droits de consultation.

```
Set<Article> set = channel.getDataSet(Article.class);
Date lastMonth = new Date(System.currentTimeMillis() - MILLIS_IN_ONE_MONTH);
DataSelector lastMonthSelector = Publication.getPdateSelector(lastMonth, null));
DataSelector visibleSelector = Publication. getVisibleStateSelector();
DataSelector noRightSelector = new ReverseSelector(Publication. getReadRightSelector());
Selector fullSelector = new AndSelector(lastMonthSelector, visibleSelector, noRightSelector);
Set<Publication> resultSet = JcmsUtil.select(set, selector, null);
```

3.3.2 Tri avec un comparateur

Il est souvent nécessaire de trier les données selon un critère particulier.

C'est par exemple le cas avec la méthode JcmsUtil.select() qui reçoit un comparateur en dernier paramètre pour ordonner l'ensemble des résultats retournés.

C'est aussi le cas avec les TreeSet qui peuvent recevoir un comparateur pour trier les données qu'ils reçoivent (à défaut les données seront triées selon leur comparateur naturel).

Les comparateurs utilisés dans JCMS doivent dériver de la classe standard Comparator.

Attention ! Pour les données stockées dans JcmsDB il est fortement recommandé de réaliser le tri des résultats sur la base de données. D'autant plus si l'affichage des résultats est paginé. Par exemple, il n'est pas efficace de charger en mémoire tous les résultats d'une recherche, de les trier en mémoire et de n'afficher que les 10 premiers.

Comparateurs existants

JCMS fournit en standard de nombreux comparateurs pour chaque type de données qu'il gère.

Tous ces comparateurs sont accessibles par la classe ComparatorManager. Pour obtenir un comparateur, il suffit d'indiquer la classe concernée et le critère de tri (généralement un nom). Il est aussi possible de d'inverser le sens naturel de la comparaison.

Pour plus de détails sur le ComparatorManager, consultez l'article *Tri de données via le ComparatorManager* (http://community.jalios.com/howto/comparator).

Le sens naturel de comparaison est propre à chaque type de comparateur. Dans JCMS, tous les comparateurs qui opèrent sur des dates, sont dans le sens ante-chronologique (du plus récent au plus ancien).

Des méthodes dépréciées existent encore pour obtenir les comparateurs auprès des classes concernées (eg Publication.getPdateComparator()) mais désormais il est déconseillé de les utiliser.

Exemples :

```
Comparator c1 = ComparatorManager.getComparator(Data.class, "cdate");
Comparator c1 = ComparatorManager.getComparator(Member.class, "name");
Comparator c1 = ComparatorManager.getComparator(Publication.class, "pdate");
Comparator c1 = ComparatorManager.getComparator(Workspace.class, "name");
```

Ecriture d'un comparateur

L'écriture d'un comparateur peut se faire en dérivant de la classe Comparator. Cependant, il est recommandé de dériver de la classe BasicComparator. Celle-ci fournit un mode de comparaison par défaut basé sur la date de création (cdate) puis l'identifiant en cas d'égalité.

Exemple : ce comparateur trie les articles selon la longueur de leur introduction.

```
public class MyArticleIntroComparator extends BasicComparator<Article> {
  public int compare(Article p1, Article p2) {
    // object nullity check
    if (p1 == null) {
      return (p2 == null) ? 0 : -1;
    }
```

```
    if (p2 == null) {
      return 1;
    }

    // Retrieve Article introduction
    String t1 = p1.getIntro(language);
    String t2 = p2.getIntro(language);
    if (t1 == null) {
      return (t2 == null) ? 0 : -1;
    }
    if (t2 == null) {
      return 1;
    }
    int length1 = t1.length();
    int length2 = t2.length();
    if (length1 < length2) {
      return -1;
    }
    if (length1 > length2) {
      return 1;
    }

    return super.compare(p1, p2);
  }
}
```

Combinaison de comparateurs

JCMS propose deux classes pour agir sur les comparateurs.

La classe ReverseCompartor renverse le sens naturel de comparaison du comparateur reçu en argument du constructeur.

La classe MultiComparator reçoit plusieurs comparateurs dans son constructeur. En cas d'égalité sur le premier, elle passe sur le second, puis au troisième, etc.

3.3.3 Filtrage par combinaison d'ensembles

Dans certains cas, il peut être pratique de combiner des ensembles de résultats fournis par les méthodes vues précédemment pour obtenir un ensemble précis de données.

Ceci est plutôt destiné aux données stockées dans JStore.

Pour cela, JCMS fournit plusieurs méthodes ensemblistes. Ces méthodes sont sûres et optimisées.

- Util.unionSet(set1, set2, …) : retourne l'union des ensembles fournis
- Util.interSet(set1, set2, …) : retourne l'intersection des ensemble fournis
- Util.subSet(set1, set2, …) : retourne le contenu de l'ensemble set1 auquel on a retiré le contenu de set2, …

Exemples :

```
Set<Category> catSet1 = pub1.getCategorySet();
Set<Category> catSet2 = pub2.getCategorySet();
Set<Category> commonCatSet = Util.interSet(catSet1, catSet2);
Set<Category> allCatSet = Util.unionSet(catSet1, catSet2);
Set<Category> onlyPub1Cat = Util.subSet(catSet1, catSet2);
```

3.3.4 Accès par une recherche multicritères

JCMS permet de rechercher des données en se basant sur plusieurs critères, dont notamment la recherche textuelle. Ce type de recherche peut être fait en utilisant des classes appelées *QueryHandler*.

Il y en a une par type de données supporté :

- QueryHandler : recherche multicritères sur les publications
- MemberQueryHandler : recherche multicritères sur les Member (stockés dans JStore)
- DBMemberQueryHandler : recherche multicritères sur les DBMember (stockés dans JcmsDB)
- AllMemberQueryHandler : recherche multicritères sur les Member et DBMember
- GroupQueryHandler : recherche textuelle sur les groupes
- WorkspaceQueryHandler : recherche textuelle sur les espaces de travail.

3.3.4.1 QueryHandler et QueryResultSet

La classe QueryHandler permet de faire une recherche multicritère portant sur des publications stockées dans JStore ou JcmsDB.

Les critères sont renseignés par des setters sur l'objet QueryHandler. Le QueryHandler permet aussi de préciser les informations de tri et de pagination. La liste des résultats est obtenue en appelant la méthode getResultSet(). Celle-ci retourne les résultats dans un objet QueryResultSet.

La classe QueryResultSet représente un ensemble de résultats d'une requête sur Publication. Elle contient non seulement les publications trouvées mais aussi leur pertinence (dans le cadre d'une recherche textuelle), des informations sur la pagination des résultats.

Exemple : rechercher les articles ou les brèves comportant le texte « foo » créés depuis 1 mois dans l'espace de travail donné.

```
Workspace ws = channel.getWorkspace(wsId);
QueryHandler qh = new QueryHandler();
qh.setMember(mbr);
qh.setText("foo");
qh.setTypes(new String[] {"Article", "SmallNews");
qh.setBeginDate(new Date(System.currentTimeMillis() - MILLIS_IN_ONE_MONTH));
qh.setDateType("cdate");
qh.setSort("cdate");
qh.setStart(0);
qh.setPageSize(10);
qh.setSearchInDB(false);
qh.setWorkspace(ws);
QueryResultSet qrs = qh.getResultSet();
SortedSet<Publication> sortedResultSet = qrs.getAsSortedSet();
```

3.3.5 MemberQueryHandler

La classe MemberQueryHandler permet de faire une recherche multicritère portant sur des Member (pas des DBMember).

Exemple : Recherche tous les membres qui s'appellent « smith » qui sont des comptes utilisateurs et qui appartiennent au groupe « externalGroup » .

```
MemberQueryHandler mqh = new MemberQueryHandler();
mqh.setText("smith");
mqh.setUsage(Member. USAGE_ACCOUNT);
```

```
mqh.setGid(externalGroup.getId());
Set<Member> resultSet = mqh.getResultSet();
```

3.3.6 DBMemberQueryHandler

La classe DBMemberQueryHandler permet de faire une recherche multicritère portant sur des DBMember.

Exemple : Recherche tous les membres qui s'appellent « smith » qui sont des comptes utilisateurs et qui appartiennent au groupe « externalGroup » .

```
DBMemberQueryHandler mqh = new DBMemberQueryHandler();
dbmqh.setText("smith");
dbmqh.setUsage(Member. USAGE_ACCOUNT);
dbmqh.setGid(externalGroup.getId());
int count = dbmqh.getCount();
List<String> idResultList = dbmqh.getIdResultList();
```

3.3.7 AllMemberQueryHandler

La classe AllMemberQueryHandler permet de faire une recherche multicritère portant à la fois sur Member et sur des DBMember.

Exemple : Recherche tous les membres qui s'appellent « smith » qui sont des comptes utilisateurs et qui appartiennent au groupe « externalGroup » .

```
AllMemberQueryHandler amqh = new AllMemberQueryHandler();
amqh.setText("smith");
amqh.setUsage(Member. USAGE_ACCOUNT);
amqh.setGid(externalGroup.getId());
List<String> idResultList = amqh.getIdResultList();
```

3.3.8 GroupQueryHandler

La classe GroupQueryHandler permet de faire une recherche textuelle portant sur des groupes.

Exemple : Recherche tous les groupes qui contiennent le texte « admin ».

```
GroupQueryHandler gqh = new GroupQueryHandler();
gqh.setGroupText("admin");
Set<Group> resultSet = gqh.getResultSet();
```

3.3.9 WorkspaceQueryHandler

La classe WorkspaceQueryHandler permet de faire une recherche textuelle portant sur des espaces de travail.

Exemple : Recherche tous les espaces qui contiennent le texte « communauté ».

```
WorkspaceQueryHandler wqh = new WorkspaceQueryHandler ();
wqh.setText("communauté");
Set<Group> resultSet = wqh.getResultSet();
```

3.3.10 Accès par une requête Hibernate

HibernateUtil

La classe HibernateUtil fournit un ensemble de méthodes statiques pour faire des requêtes portant spécifiquement sur les données stockées dans JcmsDB.

Les méthodes query() effectue une recherche sur la classe indiquée en premier paramètre en appliquant les critères de filtrage et tri fournis dans les arguments suivants. Lorsque le nombre maximum de résultats n'est pas renseigné, seuls les 1000 premiers résultats sont retournés.

Les requêtes polymorphiques sont autorisées.

Exemple :

```
// Retourne toutes les données de JcmsDB dont j_2 est l'auteur
List<Data> list = HibernateUtil.query(Data.class, "authorId", "j_2");

// Retourne toutes les recommandations dont j_2 est l'auteur, trié par date de création
croissante.
List<Recommendation> list = HibernateUtil.query(Recommendation.class, "authorId", "j_2",
"cdate asc");

// Retourne les 3 premières recommandations portant sur la publication j_300 et dont j_2 est
l'auteur, triées par date de modification décroissante
List<Recommendation> list = HibernateUtil.query(Recommendation.class,
                              new String[] {"dataId", "authorId"},
                              new String[] {"j_300", "j_2"} ,
                              "mdate desc", 0, 3, false);
```

Les méthodes queryCount() offrent les mêmes possibilités de filtrage mais retournent simplement le nombre de résultats. Elles sont donc beaucoup plus performantes.

Exemple :

```
// Retourne le nombre de recommandations du membre j_2
int count = HibernateUtil.queryCount(Recommendation.class, "authorId", "j_2");

// Retourne le nombre de recommandation portant sur la publication j_300 et dont le membre j_2
est l'auteur.
int count = HibernateUtil.queryCount(Recommendation.class,
                              new String[] {"dataId", "authorId"},
                              new String[] {"j_300", "j_2"} );
```

PublicationCriteria

Pour les requêtes portant sur les publications, il est possible de définir un ensemble de critères avec la classe PublicationCriteria. Les critères peuvent être déclarés unitairement avec les setters existants ou via un QueryHandler

Exemple :

```
QueryHandler qh = new QueryHandler();
qh.setWorkspace(workspace);
qh.setMids(author.getId());
qh.setSort("titre");
qh.setReverse(true);
qh.setCids(category.getId());
PublicationCriteria pubCriteria = new PublicationCriteria(DBFileDocument.class, qh);
int pubCount = HibernateUtil.queryCount(pubCriteria.buildCriteria(true));
List<Publication> pubList = pubCriteria.buildCriteria(false).list();
```

PageResult

Lorsque le nombre de résultats est important, il est recommandé de les paginer afin de ne pas dégrader les performances en chargeant trop d'objets de la base. Pour cela, il faut utiliser la méthode queryPublication() qui prend en paramètre un PublicationCriteria et retourne un PageResult. Un PageResult contient la tranche de résultats demandée et des informations liées à la pagination.

Exemple

```
PublicationCriteria pubCriteria = new PublicationCriteria(DBFileDocument.class);
pubCriteria.setFirstResult(20);
pubCriteria.setMaxResults(10);
PageResult<Publication> pr = HibernateUtil.queryPublication(pubCriteria);
PageResult.Status status = pr.getStatus();
if (PageResult.Status.OK == status) {
  int totalSize = pr.getTotalSize();
  List<Publication> resultList = pr.getResultList();
  System.out.println(resultList);
} else {
  System.out.println("Cannot retreive publications: " + status.getMessage(null));
}
```

3.4 Accès à l'historique des versions

JCMS propose une API permettant d'obtenir l'historique des versions d'une donnée. Il est ainsi possible de retrouver tous les états d'une donnée, y compris pour les données supprimées.

L'accès à l'historique est disponible :

- pour toutes les données stockées dans JStore
- pour les publications stockées de JcmsDB implémentant l'interface HistorizedDBData (cf. section 2.15.3).

Attention ! L'accès à l'historique des versions des données de JStore nécessite de relire le fichier store.xml. Une optimisation est faite pour ne le relire qu'à partir de l'opération de création. Cependant, l'accès à l'historique d'une donnée très ancienne sur un store de taille importante peut être long. Pour pallier à cela JCMS met dans un cache LRU les derniers historiques demandés.

Par ailleurs, le nettoyage du store affecte aussi l'historique des versions puisque certains enregistrements sont supprimés (cf. section 1.2.5).

Pour les données stockées dans JcmsDB, l'historique des versions est géré dans une table dédiée. Cette table est indexée sur les données historisées. Aussi l'accès à l'historique d'une donnée est du même ordre de grandeur que l'accès aux données.

L'accès à l'historique d'une donnée se fait par la méthode getVersionList() de la classe Channel. Elle reçoit en argument la données (ou son identifiant) et retourne une liste de données représentant chaque enregistrement de cette donnée. La liste est triée du premier au dernier enregistrement.

```
Publication pub = channel.getData(Publication.class, "j_300");
List<Publication> versionList = (List<Publication>)channel.getVersionList(pub);
System.out.println(pub + " has been saved " + versionList.size() + " times");
String firstTitle = Util.getFirst(versionList).getTitle();
System.out.println(pub + " original title was " + firstTitle);
```

Pour les données du store, cette méthode peut aussi être appelée sur une donnée supprimée.

```
String deletedPubId = "c_12345";
List<Publication> versionList = (List<Publication>)channel.getVersionList(deletedPubId);
Publication deletedPub = Util.getLast(versionList);
Date ddate = deletedPub.getDdate();
System.out.println(deletedPub + " has been saved " + versionList.size() + " times");
System.out.println(deletedPub + " has been deleted on " + ddate);
```

3.5 Accès aux données supprimés

JCMS permet de retrouver l'ensemble des données du store qui ont été supprimées. Ceci est bien entendu conditionnés aux nettoyages du store qui auront pu être effectués notamment lorsque la règle qui supprime les opérations portant sur les données supprimées est appliquée.

Attention ! l'accès aux contenus supprimé est peut être très long et consommateur en mémoire. En effet, JCMS doit pour cela relire la totalité du store et construire pour chaque données supprimées sont historique de valeur.

L'accès aux données supprimées se fait par la méthode getDeletedSet() de la classe Channel.

```
Set<Article> deletedArticleSet = (Set<Article>)channel.getDeletedSet(Article.class);
System.out.println(Util.getSize(deletedArticleSet) + " articles have been deleted.");
```

4 L’enregistrement des données

4.1 Principes

On distingue trois types d’écriture sur les données :

- Création (`create`) : ajout d’une nouvelle donnée ;
- Mise-à-jour (`update`) : modification d’une donnée existante ;
- Suppression (`delete`) : suppression d’une donnée existante.

L’API de JCMS est la même pour l’enregistrement des données de JStore et celles de JcmsDB. Néanmoins, des différences existent, notamment pour les mises à jour et pour les notifications.

Avant de faire une écriture, il est de la responsabilité du développeur de vérifier si l’écriture est possible. Une écriture peut être rejetée pour différentes raisons : le membre qui déclenche l’écriture n’a pas les droits, la donnée est incomplète, les écritures sont arrêtées, …

La classe Data dispose des méthodes checkCreate(), checkUpdate() et checkDelete() pour vérifier chaque type d’écriture. La méthode reçoit en paramètre le membre qui effectue l’écriture. Le résultat de l’appel est un objet de la classe ControllerStatus. L’invocation de la méthode isOK() retourne true si l’écriture est possible, sinon le ControllerStatus fournit des informations sur les raisons du rejet.

4.2 Créer une donnée

Une fois la vérification effectuée, la création d’une nouvelle donnée se fait en appelant la méthode performCreate() sur l’objet. La méthode performCreate() reçoit en paramètre le membre qui effectue l’opération. Ce paramètre n’est pas l’auteur de la donnée mais l’auteur de l’écriture. Une écriture peut être effectuée sur une donnée par un membre qui n’en est pas l’auteur. Ce paramètre sert donc à tracer les auteurs des écritures pour les données historisées (i.e. toutes les données de JStore et les données de JcmsDB qui dérivent de la classe HistorizedDBData).

```
Member opAuthor = channel.getMember("… ");
Member group = new Group() ;
group.setName("Jedi Council") ;

// Chek if the group can be created by opAuthor
ControllerStatus status = group.checkCreate(opAuthor);
if (status.isOK()) {
  // Create the group
  group.performCreate(opAuthor);
} else {
  String msg = status.getMessage(opAuthor.getLanguage());
```

```
  System.out.println("Cannot create group " + group + ": " + msg);
}
```

4.3 Mettre à jour une donnée

La mise à jour d'une donnée se fait en modifiant un objet représentant la donnée puis en appelant la méthode performUpdate(). Comme pour performCreate(), elle reçoit en paramètre l'auteur de l'opération. L'objet sur lequel est déclenchée la mise à jour diffère selon qu'il s'agit de JStore ou de JcmsDB. Dans le cas de JStore, il faut faire la mise à jour sur une copie de la donnée. Dans le cas de JcmsDB, la mise à jour doit s'opérer directement sur la donnée. Pour atténuer cette différence, la méthode getUpdateInstance() renvoie l'instance sur laquelle il faut effectuer la mise à jour. Attention ! dans cas de JStore, il s'agit d'une copie de surface (« shallow copy »). Aussi, si des attributs de type tableau, collections, map, … doivent être mis à jour, il faut préalablement en faire une copie.

```
Member opAuthor = channel.getMember("…");
Group group = channel.getGroup(groupId);
Group aNewParent = channel.getGroup("…");

// Get a copy of the group to update it
Group updated = (Group)group.getUpdateInstance();

// Update the name
updated.setName("The Jedi Council");


// Add a new parent. Make a copy of parentSet
HashSet<Group> newParentSet = new HashSet<Group>(updated.getParentSet());
newParentSet.add(aNewParent);
updated.setParentSet(newParentSet);

// Check if the group can be updated
ControllerStatus status = updated.checkUpdate(opAuthor) ;
if (status.isOK()) {
   // Update the group
   updated.performUpdate(opAuthor);
} else {
  String msg = status.getMessage(opAuthor.getLanguage());
  System.out.println("Cannot update group " + group + ": " + msg);
}
```

Attention ! Dans le cas de JcmsDB, une fois la donnée modifiée, elle sera mise à jour même si la méthode performUpdate() n'est pas appelée. Il faut appeler explicitement la méthode HibernateUtil.evict(data) pour annuler les modifications sur cette donnée. Cet appel est automatiquement réalisé par le FormHandler lorsque la méthode checkUpdate() rejette la mise à jour. Pour les appels en dehors du FormHandler, c'est au développeur de déclencher l'invocation de la méthode evict().

4.4 Supprimer une donnée

La suppression d'une donnée se fait en appelant la méthode performDelete() sur la donnée.

```
Member opAuthor = channel.getMember("…");
Group group = channel.getGroup(groupId);

// Check if the group can be deleted
ControllerStatus status = group.checkDelete(opAuthor);
if (status.isOK()) {
  // Delete the group
  group.performDelete(opAuthor);
} else {
  String msg = status.getMessage(opAuthor.getLanguage());
  System.out.println("Cannot delete group " + group + ": " + msg);
}
```

4.5 Verrouiller une donnée

L'API de JCMS permet de verrouiller une donnée. Le verrouillage sert à prévenir voire empêcher un autre utilisateur de modifier ou supprimer une donnée.

JCMS propose deux types de verrouillage :

- Le verrouillage souple : verrouillage implicite et stocké en mémoire pour une durée limitée
- Le verrouillage dur : verrouillage explicite et persisté dans JcmsDB sans limite de temps.

4.5.1 Verrouillage souple

Le verrouillage souple est implicite dès qu'un utilisateur édite une donnée avec les interfaces Web de JCMS. Si un autre utilisateur tente d'éditer cette donnée avant que l'édition soit terminée, un message le prévient qu'un autre utilisateur édite actuellement cette donnée.

L'API de JCMS fournit les méthodes pour connaître les informations sur les verrous souples :

```
Publication pub = channel.getPublication(pubId);
if (pub.isLocked()) {
  Member lockMbr = pub.getLockMember();
  Date lockDate = pub.getLockDate() ;
  System.out.println(pub + " has been locked by " + lockMbr + " since "  + lockDate) ;
}
```

Attention ! le verrouillage souple étant géré uniquement en mémoire, dans un cluster JSync, il n'est partagé qu'avec les contributeurs du même réplica.

4.5.2 Verrouillage dur

Le verrouillage dur est explicite et persistant. Il n'opère que sur les publications. Lorsqu'une publication a été verrouillée ainsi, seul le membre qui a posé le verrou peut la mettre à jour ou la supprimer. L'interface de JCMS permet cependant aux autres contributeurs de demander le relâchement et à l'administrateur de forcer le relâchement.

```
Publication pub = channel.getPublication(pubId);
Member mbr = channel.getMember(mbrId);
pub.putStrongLock(mbr);
Member lockMbr = pub.getStrongLockMember();
Date lockDate = pub.getStrongLockDate();
System.out.println(pub + " has been locked by "  + lockMbr + " since " + lockDate);
pub.releaseStrongLock(mbr);
```

5 Intervenir dans le traitement d’un enregistrement

5.1 DataController

Les DataController gèrent l’intégrité des données et interviennent dans le traitement des écritures de toutes les données JCMS.

Toutes les classes de données de JCMS intègrent nativement un contrôle d’intégrité (p. ex. pour vérifier les champs obligatoires de la donnée).

Attention ! dans un cluster JSync, les DataController ne sont appelés que sur le réplica sur lequel l’écriture à lieu. Pour déclencher un traitement sur l’ensemble des réplicas, il faut utiliser un StoreListener ou un DBListener.

La déclaration d’un nouveau DataController se fait via un fichier plugin.xml. L’attribut types indique sur quels types de données le DataController doit être appelé.

```
<plugincomponents>
  ...
  <datacontroller class="com.example.MyDataController"
                  types="Member Article PortletSearch" />
  ...
</plugincomponents>
```

5.1.1 Validation et contrôle d'intégrité

L’API de JCMS permet de développer de nouveaux DataController pour tous les types de données JCMS. Il est ainsi possible d’ajouter de nouvelles règles d’intégrité sur une donnée. Par exemple, sur un contenu Meeting comportant un champ « startDate « et un champ « endDate », vérifier que la startDate est bien antérieure à la endDate. Autre exemple, sur un membre, vérifier que le mot de passe comporte au moins 8 caractères dont un chiffre.

Le contrôle d’intégrité est réalisé en dérivant de la classe BasicDataController et en surchargeant la méthode checkIntegrity().

```
public class MeetingDataController extends BasicDataController  {
  @Override
  public ControllerStatus checkIntegrity(Data data) {
    Meeting meeting = (Meeting)data;
    long startTime = meeting.getStartDate().getTime();
    long endTime = meeting.getEndDate().getTime();
    if (startTime < endTime) {
      ControllerStatus status = new ControllerStatus();
      status.setMessage("Start date must be before end date");
      return status;
    }
    return ControllerStatus.OK;
```

```
  }
}
```

Le contrôle d'intégrité est appelé durant la phase de contrôle de l'écriture (lors de l'appel des méthode checkCreate(), checkUpdate()) mais aussi dans de nombreux points de l'interface de JCMS (p. ex. dans la liste des contenus). Il est donc impératif que le contrôle d'intégrité soit performant. Si ce n'est pas possible il faut déporter le traitement dans le contrôle de l'écriture.

Par ailleurs, un DataController ne peut qu'ajouter de nouvelles contraintes d'intégrité ; il ne peut pas en supprimer.

5.1.2 Contrôle de l'exécution des écritures

Les DataController permettent aussi d'agir en amont et en aval d'une écriture. Ils interviennent dans la phase de contrôle d'une écriture, en plus du contrôle d'intégrité. Le contrôle d'intégrité rejette une écriture pour des raisons propres à la donnée. Le contrôle d'une écriture la rejette pour des raisons externes à la donnée. Par exemple : le membre qui effectue l'écriture, l'état de JCMS au moment de l'écriture, la date de l'écriture, … Comme le contrôle de l'écriture n'a lieu que lorsqu'une écriture est déclenchée, il est moins critique sur les performances que le contrôle d'intégrité.

Le contrôle de l'écriture est réalisé en dérivant de la classe BasicDataController et en surchargeant la méthode checkWrite().

```
public class PasswordController extends BasicDataController {
  @Override
  public ControllerStatus checkWrite(Data data, int op, Member mbr, boolean checkIntegrity,
Map context) {

    // Control only creation and update
    if (op == OP_DELETE) {
      return ControllerStatus.OK;
    }

    // Get the request to extract the uncrypted password
    Channel channel = Channel.getChannel();
    HttpServletRequest request = (HttpServletRequest)channel.getCurrentServletRequest();
    if (request == null) {
      return ControllerStatus.OK;
    }
    String password = request.getParameter("password1");
    if (Util.isEmpty(password)) {
      return ControllerStatus.OK;
    }

    // Check the password
    if (checkPassword(password)) {
      return ControllerStatus.OK;
    }

    return new ControllerStatus("The password must contains between 8 and 16 characters and at
least 1 digit.");
  }

  private boolean checkPassword(String password) {

    // At least 8 characters (max 16)
    if (password.length() < 8 || password.length() > 16) {
      return false;
    }

    // At least 1 digit
    for(int i = 0; i < password.length(); i++) {
      if (Character.isDigit(password.charAt(i))) {
        return true;
      }
    }
```

```
    return false;
  }
}
```

Une fois la phase de contrôle passée, les DataController interviennent juste avant et juste après l'écriture. Ils peuvent ainsi modifier l'objet juste avant qu'il ne soit enregistré ou déclencher un traitement juste après l'écriture. Par exemple, il est possible de signer une publication et d'ajouter la signature dans un champ dédié, de faire une catégorisation automatique de la publication selon son contenu, …

Attention ! durant cette phase les modifications ne sont plus contrôlées. C'est au développeur de garantir qu'elles respectent bien les contraintes d'intégrité de l'objet. Par ailleurs, plusieurs DataController peuvent intervenir lors d'une écriture. Il est possible de donner une indication sur l'ordre d'appel mais on ne peut pas garantir l'ordre d'appel de deux DataController qui ne se connaissent pas. Aussi, les DataController doivent prendre en compte qu'un autre DataController a pu être appelé avant et a pu modifier la donnée.

Les opérations avant et après l'écriture sont réalisées en dérivant de la classe BasicDataController et en surchargeant les méthodes beforeWrite() et afterWrite().

```
public class DigestController extends BasicDataController {

  @Override
  public void beforeWrite(Data data, int op, Member mbr, Map context) {

    if (op == OP_DELETE) {
      return;
    }

    ArticlePhoto pub = (ArticlePhoto)data;
    pub.setDigest(getDigest(pub));
  }

  String getDigest(Publication pub) {
    // compute the signature…
  }
}
```

Pour plus de détails sur les DataController, consultez l'article *Développer avec DCM et les DataController* (http://community.jalios.com/howto/datacontroller).

5.2 StoreListener

Les StoreListener écoutent les écritures des données dans JStore. Contrairement au DataController, ils ne peuvent pas agir sur ces écritures mais juste déclencher un traitement à posteriori.

Les StoreListener sont appelés non seulement lorsque l'écriture a lieu mais aussi :

- au démarrage de JCMS lors du chargement du Store
- dans un cluster JSync, lorsqu'une nouvelle écriture arrive sur un réplica

Les StoreListener sont donc bien adaptés pour gérer des index sur les données du Store ou pour déclencher une opération sur un réplica particulier (p. ex. le leader).

Attention ! JCMS utilise des StoreListener interne pour gère les indexes sur données. L'ordre d'appel des StoreListener n'est pas déterministe. Votre StoreListener peut être appelé avant ou après ceux

de JCMS. Aussi, votre StoreListener ne doit pas tester l'existence de la données qui vient d'être créé par les API de JCMS car les indexes n'ont peut être pas encore été mis à jour.

La déclaration d'un nouveau StoreListener se fait via un fichier plugin.xml. L'attribut types indique pour quels types de données le StoreListener doit être appelé. L'attribut beforeStoreLoad indique si le StoreListener doit être appelé lors du chargement du Store. L'attribut repeat indique s'il doit être appelé lorsqu'une écriture est rejouée dans un cluster JSync.

```
<plugincomponents>
  ...
  <storelistener class="com.example.MyStoreListener"
                 types="Member Article PortletSearch"
                 beforeStoreLoad="false"
                 repeat="true" />
  ...
</plugincomponents>
```

Pour implémenter un StoreListener il faut dériver de la classe [BasicStoreListener]() et surcharger les méthodes handleCreate(),handleCommitUpdate() et handleDelete(). L'argument firstTime indique si l'opération est jouée pour la première fois qu'elle a eu lieu, c'est-à-dire qu'elle n'est pas jouée lors du chargement du store ni lors de la diffusion dans un cluster JSync.

Sur un cluster JSync, il est parfois nécessaire de ne déclencher un traitement qu'une seule fois pour tout le cluster (p. ex. l'envoi d'un mail, ou un échange avec un système tiers). Une solution consiste à ne déclencher le traitement que sur le leader. Pour cela, il suffit d'appeler la méthode channel.isMainLeader() qui retourne true uniquement pour le leader du cluster.

Dans l'exemple suivant, on affiche un message dans la console du leader lorsqu'un nouveau membre avec les droits administrateur est créé ou supprimé.

```
public class MemberListener extends BasicStoreListener {

  @Override
  public void handleCreate(Storable storable, boolean firstTime) {

    if (!isLeader()) {
      return;
    }
    Member mbr = (Member)storable ;
    if (!mbr.isAdmin()) {
      return ;
    }
    System.out.println("A new admin has been created : "  + mbr) ;
  }

  @Override
  public void handleDelete(Storable storable, boolean firstTime) {
    if (!isLeader ()) {
      return;
    }
    Member mbr = (Member)storable ;
    if (!mbr.isAdmin()) {
      return ;

    }
    System.out.println(" Admin "  + mbr + " has been deleted") ;
  }



  private boolean isLeader() {
    return Channel.getChannel().isMainLeader();
```

```
  }
}
```

5.3 DBListener

Les DBListener sont le pendant des StoreListener pour JcmsDB. Ils écoutent les écritures qui ont lieu sur les données stockées en base.

Dans un cluster JSync, les DBListener sont déclenchés sur les différents réplicas mais avec un retard par rapport au réplica par lequel a eu lieu l'écriture. Contrairement aux DataController et à l'instar des StoreListener, les DBListener permettent de rejouer un traitement consécutif à une écriture sur l'ensemble du cluster.

Les DBListener sont donc bien adaptés pour gérer des index sur les données de JcmsDB ou pour déclencher une opération sur un réplica particulier (p. ex. le leader). Attention cependant à ne pas garder de référence sur la donnée mais uniquement sur son identifiant (afin de ne pas empêcher le GC de libérer les données de JcmsDB chargée en mémoire temporairement).

La déclaration d'un nouveau DBListener se fait via un fichier plugin.xml. L'attribut types indique pour quels types de données le DBListener doit être appelé.

```
<plugincomponents>
  ...
  <dblistener class="com.example.MyDBListener" types="DBFileDocument" />
  ...
</plugincomponents>
```

Pour implémenter un DBListener il faut dériver de la classe BasicDBListener et surcharger les méthodes onPostInsert(), onPreUpdate(), onPostUpdate() et onPostDelete(). Ces méthodes recoivent la donnée, l'argument event contenant les informations sur l'écriture et l'argument firstTime qui indique si l'écriture est jouée pour la première fois dans le cluster.

Dans l'exemple suivant, on affiche un message dans la console du leader lorsqu'un nouveau compte DBMembre est créé ou supprimé.

```
public class DBMemberListener extends BasicDBListener {

  @Override
  public void onPostInsert(DBData data, PostInsertEvent event, boolean firstTime) {

    if (!isLeader()) {
      return;
    }
    DBMember mbr = (DBMember)data;
    if (!mbr.isAccount()) {
      return ;
    }
    System.out.println("A new DBMember account has been created:"  + mbr) ;
  }

  @Override
  public void onPostDelete (DBData data, PostDeleteEvent event, boolean firstTime) {
    if (!isLeader ()) {
      return;
    }
    Member mbr = (Member)data ;
    if (!mbr.isAccount ()) {
      return ;
    }
    System.out.println("DBMember account "  + mbr + " has been deleted") ;
  }
```

```
  private boolean isLeader() {
    return Channel.getChannel().isMainLeader();

  }
}
```

Lorsqu'il est nécessaire d'écouter à la fois des écritures sur JStore et sur JcmsDB, il est possible d'utiliser la classe BasicDBListenerStoreListener qui implémente par défaut les méthodes des deux interfaces.

6 Le contrôle des droits d'accès

JCMS offre en standard une large palette de droits sur l'ensemble des fonctionnalités : droits de consultation, de contribution, de dépôt de fichier, d'administration, …

Dans JCMS les droits sont affectés soit individuellement à des membres soit à des groupes. JCMS permet de faire des groupes de groupes. Un membre qui appartient à un sous-groupe appartient automatiquement à tous les groupes parents de ce groupe (et aux parents des parents). Ainsi en affectant un droit à un groupe c'est non seulement les membres de ce groupe qui en bénéficient mais aussi tous les membres des sous-groupes de ce groupe.

Pour certains besoins il peut être nécessaire de spécialiser la gestion de droit. La spécialisation peut se faire soit en complément soit en remplacement de l'implémentation standard. Ce principe est très puissant puisqu'il permet de faire des droits dynamiques voire contextuels à la requête. En contrepartie, ceci impose de toujours faire les contrôles de droits au moment où les données sont accédées.

6.1 Droits de consultation

6.1.1 Publication

Ce droit contrôle qui peut consulter un contenu ou visualiser une portlet. Ce droit est défini contenu par contenu en indiquant les groupes et les membres autorisés. Si aucun droit n'est défini, la publication est visible de tous. Il est possible d'affecter des droits de consultation par défaut (par type de publication ou par espace de travail).

Le droit de consultation est conditionné :

- à l'état du workflow dans lequel se trouve la publication : une publication hors de l'état publié n'est visible que dans le back-office de son espace de travail.
- au rédacteur : quelques soient les droits de consultation, un membre a toujours accès aux contenus dont il est le rédacteur.

La méthode canBeReadBy() de la classe Publication et la méthode canRead() de la classe Member servent à tester ce droit.

```
Publication pub = channel.getPublication(pubId) ;
Member mbr = channel.getMember(mbrId) ;
if (pub != null && pub.canBeReadBy(mbr)) {
```

```
  System.out.println(mbr + " can read publication " + pub) ;
}
```

Ce code est équivalent à :

```
Publication pub = channel.getPublication(pubId) ;
Member mbr = channel.getMember(mbrId) ;
if (mbr != null && mbr.canRead(pub)) {
  System.out.println(mbr + " can read publication "  + pub) ;
}
```

La classe Channel fournit aussi la méthode getPublicationSet() qui permet de retourner un ensemble de publications d'un certain type qui sont accessibles pour un membre donné.

```
Member mbr = channel.getMember(mbrId) ;
TreeSet<Publication> pubSet = channel.getPublicationSet(Article.class, mbr);
```

6.1.2 Catégorie

Comme pour les publications, des droits de consultation peuvent être appliqués aux catégories.

La méthode canBeReadBy() de la classe Category et la méthode canRead() de la classe Member servent à tester ce droit.

```
Category cat = channel.getCategory(catId) ;
Member mbr = channel.getMember(mbrId) ;
if (cat != null && cat.canBeReadBy(mbr)) {
  System.out.println(mbr + " can read category "  + cat) ;
}
```

6.1.3 Groupe

Comme pour les publications, des droits de consultation peuvent être appliqués aux groupes. Par défaut, JCMS propose 3 niveaux de visibilité de groupes :

1. Le groupe est visible de tout le monde
2. Le groupe n'est visible que des membres qui en font partie
3. Le groupe n'est visible que des administrateurs.

Cependant, grâce à la spécialisation des droits il est possible d'étendre ces règles.

La méthode canBeReadBy() de la classe Group et la méthode canRead() de la classe Member servent à tester ce droit.

```
Group grp = channel.getGroup(grpId) ;
Member mbr = channel.getMember(mbrId) ;
if (grp != null && grp.canBeReadBy(mbr)) {
  System.out.println(mbr + " can view group "  + group) ;
}
```

6.2 Droits de contribution

Ce droit contrôle qui peut créer, modifier ou supprimer des publications. Ce droit est défini par type de donnée. Il est affecté pour un groupe ou pour un membre.

Ce droit est applicable sur les types Publication et Category.

Pour les publications plusieurs méthodes permettent de contrôler les droits de contribution.

6.2.1 Member.canPublish()

La méthode canPublish(class,workspace) indique si un membre peut publier une publication d'une classe donnée dans un espace de travail donné.

```
Member mbr = channel.getMember(mbrId) ;
Workspace ws = channel.getWorkspace(wsId) ;
if (mbr.canPublish(Article.class, ws)) {
  System.out.println(mbr + " can publish new article in workspace "  + ws) ;
}
```

6.2.2 Member.canPublishSome()

La méthode canPublishSome(class,workspace) indique si un membre peut publier une publication d'une classe donnée ou d'une de ses sous-classes dans un espace de travail donné.

```
Member mbr = channel.getMember(mbrId) ;
Workspace ws = channel.getWorkspace(wsId) ;
if (mbr.canPublishSome(FileDocument.class, ws)) {
  System.out.println(mbr + " can publish FileDocument in workspace "  + ws) ;
}
```

6.2.3 Member.canWorkOn()

La méthode canWorkOn() de la classe Member indique si un membre peut travailler sur une donnée dans son état actuel.

```
Member mbr = channel.getMember(mbrId) ;
Publication pub = channel.getPublication(pubId) ;
if (mbr.canWorkOn(pub)) {
  System.out.println(mbr + " can edit publication " + pub) ;
}
```

La méthode peut recevoir en paramètre un RightInfo qui fournit en cas de refus une explication.

```
Member mbr = channel.getMember(mbrId) ;

Publication pub = channel.getPublication(pubId) ;
RightInfo rightInfo = new RightInfo() ;
if (!mbr.canWorkOn(pub, rightInfo)) {
  String explanation = rightInfo.getExplanationMessage(userLang, userLocale, pub, mbr);
  System.out.println(mbr + " cannot edit publication "  + pub + " because " + explanation) ;
}
```

6.2.4 Member.canManageCategory()

La méthode canManageCategory() de la classe Member indique si un membre peut éditer, supprimer, déplacer et créer des sous-catégories dans une catégorie donnée.

```
Member mbr = channel.getMember(mbrId) ;

Category cat = channel.getCategory(catId) ;
if (mbr.canManageCategory(cat)) {
  System.out.println(mbr + " can edit, delete, move and add descendants in category "  +
cat) ;
}
```

6.2.5 Data.checkCreate(), checkUpdate(), checkDelete()

Les méthodes checkCreate(), checkUpdate() et checkDelete() de la classe Data indiquent si un membre peut réaliser l'écriture sur une donnée. Ces méthodes testent notamment les droits de contribution. Mais elles vont au-delà des méthodes précédentes car elles vérifient si toutes les conditions sont réunies actuellement pour que l'écriture soit validée (p. ex. elles vérifient si les écritures sont activées).

Voir la section 3.5 pour plus de détail sur ces méthodes.

6.3 ACL

Les ACL (Access Control List, ou liste de contrôle d'accès) permettent d'attribuer des droits d'administration aux utilisateurs, par fonctionnalité, pour l'administration centrale ou pour l'administration d'un ou plusieurs espaces de travail.

Chaque ACL permet de spécifier des fonctionnalités autorisées à être utilisées (autrement appelées ressources).

Une ACL est par la suite référencée par un ou plusieurs groupes. Tous les membres de ce groupe (et de ses sous-groupes) bénéficieront des droits d'accès définis dans l'ACL. Plusieurs ACL peuvent être créées sur le site.

6.3.1 Types d'ACL

On distingue 2 types d'ACL :

- les ACL globales, qui permettent d'autoriser l'accès à des fonctionnalités d'administration centrale,
  elles ne peuvent être rattachées qu'à des groupes globaux, uniquement par un administrateur central
- les ACL d'espace de travail, qui permettent d'autoriser l'accès à des fonctionnalités d'administration d'espace de travail,
  elles ne peuvent être rattachées qu'à des groupes d'espace de travail (par un administrateur central ou un administrateur d'espace de travail)

6.3.2 Déclaration des ACL

Le système des ACL est extensible. Il est possible d'ajouter de nouvelles ressources dont l'accès sera contrôlé par les ACL. La définition des ressources qui peuvent être utilisées dans une ACL s'effectue via les propriétés de JCMS. Chaque ressource est représentée sous la forme d'un chemin similaire à une URI ou un chemin de fichier Unix.

Lorsque la ressource concerne une fonctionnalité d'administration centrale, elle est préfixée par `admin/`, lorsqu'il s'agit d'une fonctionnalité d'administration d'espace de travail, elle est préfixée par `admin-ws/`.

La déclaration d'une ressource se fait via une propriété de configuration (vide) et les propriétés de traduction correspondantes. Toutes ces propriétés respectant la même nomenclature :

```
acl.resource.{chemin-de-la-ressource}.
```

Par exemple, la fonctionnalité permettant la consultation du *Journal des évènements* de JCMS est déclarée comme ceci :

- Dans `jcms.prop`
- `acl.resource.admin:`
- `acl.resource.admin/monitoring:`
  `acl.resource.admin/monitoring/logs:`

- Dans `fr.prop`
- `acl.resource.admin: Espace d'administration`

- `acl.resource.admin/monitoring: Supervision`
- `acl.resource.admin/monitoring/logs: Journal des évènements`

Cette représentation sous forme arborescente offre les caractéristiques suivantes :

- le chemin de la ressource est en correspondance avec la navigation permettant d'accéder à la fonctionnalité dans l'interface graphique
  exemple : l'accès au "Journal des évènements" est situé dans la zone "Supervision", elle-même se trouvant dans l'espace d'administration
- le droit d'accès à une ressource fille donne accès à tous ses parents
  exemple : l'accès au "Journal des évènements" donne accès à la zone d'administration
- le droit d'accès à une ressource parente donne le droit à toutes les ressources filles
  exemple : l'accès à la zone "Supervision" donne accès au "Journal des évènements"
- le chemin d'une ressource est sans aucun lien avec le chemin "physique" du jsp/servlet dans l'arborescence de la webapp JCMS

6.3.3 Vérification des ACL

La vérification des droits d'accès à une fonctionnalité d'administration s'effectue via la méthode `checkAccess()` recevant en paramètre le chemin de la ressource :

```
if (!checkAccess("{resource-path}")) {
  sendForbidden(request, response);
  return;
}
```

Par exemple, l'accès à fonctionnalité "Journal des évènements" est contrôlé de la façon suivante :

```
if (!checkAccess("admin/monitoring/logs")) {
  sendForbidden(request, response);
  return;
}
```

Cette méthode contrôle automatiquement que les droits d'accès du membre connecté lui permettent d'accéder à la ressource demandée, via une ACL d'un groupe auquel il appartient ou via des droits d'administration habituels (administrateur d'espace ou administrateur central).

6.4 Autres droits

JCMS dispose de nombreux autres droits sur diverses fonctionnalités : droits d’archiver, droit de fusionner une copie de travail, droit de déposer une photo, droit d’utiliser une branche de catégorie, …

Consultez la JavaDoc de la classe Member pour avoir plus d’information sur ces différentes méthodes.

6.5 Spécialisation des droits

La spécialisation des droits permet d’enrichir le comportement natif de JCMS. La spécialisation peut se faire soit en complément soit en remplacement de l'implémentation standard.

Il est possible d'adapter ou de modifier une partie du système de droits de JCMS en ajoutant un module comportant des composants RightPolicyFilter. Chacune des méthodes de cette interface permet de surcharger le comportement d'une famille de droit de JCMS.

L'ajout d'un RightPolicyFilter se fait dans le fichier plugin.xml du module via la balise <policyfilter>.

Exemple de déclaration :

```
<plugincomponents>
  ...
  <policyfilter class="org.demo.jcmsplugin.demo.MyRightPolicyFilter" />
  ...
</plugincomponents>
```

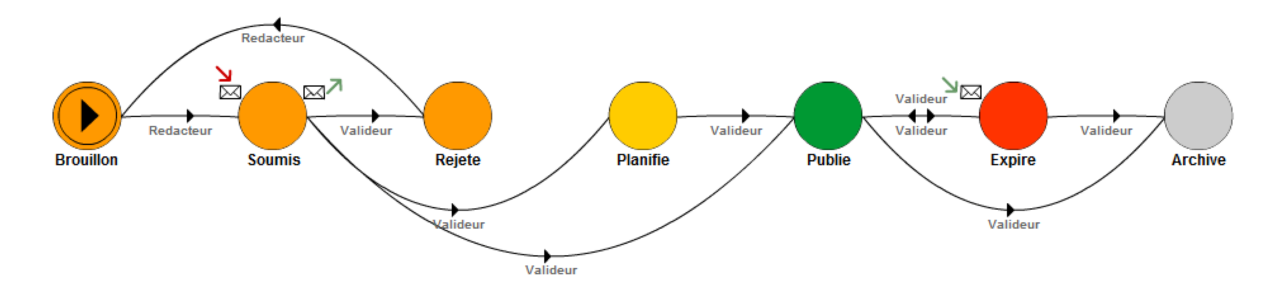
Pour plus de détails consultez l’article *Personnalisation de la gestion des droits* (http://community.jalios.com/howto/rightpolicy).

7 La gestion des Workflows

Toute publication JCMS est soumise à un workflow. Par défaut, il s'agit du workflow minimal qui comporte 4 états :

- Planifié
- Publié
- Expiré
- Archivé

Un workflow est d'un ensemble d'états reliés entre eux par un ensemble de transition. Chaque transition est étiquetée par un rôle. Le workflow comporte un état initial par lequel début le traitement.



JCMS propose plusieurs modèles de Workflow et permet d'en définir des nouveaux. Les modèles de workflow se répartissent en deux types :

- Les workflow de publication : destinés à des process éditoriaux
- Les workflow de traitement : destinés à des process de traitement

Dans les workflow de publication, les 4 états Planifié, Publié, Expiré et Archivé sont obligatoires.

Dans les workflow de traitement, seul l'état Archivé est obligatoire.

Les workflows sont sauvegardé dans un fichier XML situé dans le répertoire `WEB-INF/data/workflows/`.

Un module peut livrer ses propres workflow en les déclarant dans le fichier plugin.xml avec la balise `<workflows>` :

```
<plugin …>
```

```
  …
  <workflows>
    <workflow id="myWorkflow" />
  </workflow>
  …
</plugin>
```

Les Workflows sont des types de données de JCMS qui ne sont pas dans l'arborescence Data.

Dans l'API JCMS, les workflows sont manipulés avec les classes suivantes :

- Workflow
- WFState
- WFAction
- WFTransation
- WFRole
- WKRole

7.1 La classe WorkflowManager

La classe WorkflowManager fournit des méthodes pour accéder aux workflows.

Les principales méthodes

Méthode	Description
getDefaultWorkflow()	Retourne le workflow par défaut (avec uniquement les 4 états par défaut).
getWorkflow(wfId)	Retourne le workflow d'identifiant wfId.
getWorkflowSet()	Retourne l'ensemble des workflows.

7.2 La classe Workflow

La classe Workflow représente un Workflow. Elle permet d'accéder à ses attributs, ses états, ses transitions et ses rôles.

7.2.1 Les principaux attributs

Attribut	Description
Id	Identifiant du Workflow
labelMap	Map des libellés localisés
Type	Type de workflow
initState	Etat initial

7.2.2 Les principales méthodes

Méthode	Description
getLabel(userLang)	Retourne le libellé du workflow dans la langue indiquée.
getStateSet()	Retourne l'ensemble des états du workflow
getTransitionSet()	Retourne l'ensemble des transitions du workflow
getRoleSet()	Retourne l'ensemble des rôles du workflow
canChangeState(mbr, pub, source, target)	Retourne true si le membre mbr peut passer la publication pub de l'état source à l'état target
getNextStateSet(pub, mbr)	Retourne tous les états atteignables par le membre mbr pour la publication pub.

canWorkInState(pub, mbr, state)	Retourne true si le membre mbr peut travailler dans l'état state pour la publication pub.

7.3 La classe WFState

La classe WFState représente un état d'un Workflow.

Un état est identifié par son pstatus. Il s'agit d'une valeur numérique avec les propriétés suivantes :

- Le pstatus 0 identifie l'état Publié. Cet état est visible
- Les pstatus pairs représentent les états invisibles
- Les pstatus impairs représentent les états visibles

Lorsqu'une publication est dans un état invisible, elle est visible en back-office mais elle n'est plus visible en front-office.

Il est possible de définir une règle de relance sur un état. Dans ce cas, lors la durée de la relance est atteinte, les membres pouvant agir sur les publications dans cet état sont relancé.

Il est aussi possible de définir une règle de transition automatique au bout d'un certain temps. C'est la fonction appelé Workflow Express.

Des actions peuvent être associées lorsqu'une publication rentre ou sort d'un état. Par ces actions il est possible d'envoyer une alerte :

- A l'auteur de la publication
- Au membre pouvant agir sur la publication
- A l'administrateur de l'espace de travail dans lequel est la publication

Les principales méthodes

Méthode	**Description**
getPsatus()	Retourne la valeur associée à cet état.
getReminder()	Retourne la durée de relance (0 = pas de relance)
getDuration()	Retourne la durée du Workflow Express
getTargetPstatus()	Retourne l'état cible pour le Workflow Express
getTransitionSet()	Retourne l'ensemble des transitions sortant de cet état.
getRoleSet()	Retourne l'ensemble des rôles associés aux transitions de cet état
hasReminder()	Retourne true si une relance est associée à cet état
isExpress()	Retourne true si un Workflow Express est associée à cet état
isSpecialState()	Retourne true si il s'agit de l'un des 4 états spéciaux (Planifié, Publié, Expiré, Archivé)
getActionOutSet()	Retourne l'ensemble des actions sortantes
getActionInSet()	Retourne l'ensemble des actions entrantes

7.4 La classe WFAction

La classe WFAction est la super classe des différentes actions qui peuvent être déclenchée à l'entrée ou à la sortie d'un état.

7.5 La classe WFTransition

La classe WFTransition représente une transition entre deux états d'un Workflow.

Les principales méthodes

Méthode	Description
getSource()	Retourne l'état de départ
getTarget()	Retourne l'état d'arrivé
getRoleSet()	Retourne l'ensemble des rôles de cette transition.

7.6 La classe WFRole

La classe WFRole représente un rôle dans le workflow. Les rôles sont des notions abstraites qui sont concrétisés au sein des espaces de travail.

Par exemple le workflow Basique comporte le rôle Valideur. Celui-ci est utilisé pour passer la publication de l'état Soumis à l'état Planifié ou Publié.

Le choix des membres ou des groupes qui doivent être associés à ce rôle est défini pour chaque espace de travail utilisant ce Workflow. Cette association est matérialisée par la classe WKRole.

Les principales méthodes

Méthode	Description
getId()	Retourne l'identifiant du rôle. A partir de JCMS 9, cette méthode retourne la concaténation de l'identifiant du workflow et de l'attribut roleId (p. ex. basic-writers)
getName(lang)	Retourne le nom du rôle dans la langue indiquée.
getDescription(lang)	Retourne la description du rôle dans la langue indiquée.

7.7 La classe WKRole

La classe WKRole représente la définition d'un rôle dans un espace de travail.

Un rôle comporte :

- La liste de membres qui ont le rôle
- Un mode de comportement sur les membres (au moins un membre ou tous les membres doivent avoir validés)
- La liste des groupes pour lesquels les membres ont le rôle
- Un map de pondération pour chaque groupe (p. ex. 51% des membres du groupe)
- Un mode de comportement sur les groupes (au moins un groupe ou tous les groupes pondérés doivent avoir validés)
- Est-ce qu'il s'agit d'un rôle ouvert (un rôle ouvert permet au membre qui édite la publication de choisir les membres jouant le rôle sur cette publication) ?

Les principales méthodes

Méthode	Description
isInvolved(pub, mbr, boolean)	Retourne true si le membre mbr participe à ce WKRole sur la publication pub.
getWorkflow()	Retourne le Workflow associé à ce WKRole
getWFRole()	Retourne le WFRole associé à ce WKRole

7.8 La classe Publication

Les classes dérivant de la classe Publication disposent de méthodes pour obtenir des informations sur le workflow.

Les principales méthodes

Méthode	Description
assignRole(roleId, mbr)	Affecte un membre à un rôle.
assignRole(roleId, mbrSet)	Affecte un ensemble de membres au rôle indiqué.
getMemberSetAssignedToRole(roleId)	Retourne l'ensemble des membres associé au rôle indiqué.
getNextWFStateSet(mbr)	Retourne l'ensemble des WFState dans lesquels le membre indiqué peut placer la publication (en fonction de ses droits, du workflow et de l'état courant de la publication)
getPstatus()	Retourne l'état de la publication
getRoleMap()	Retourne la Map des rôles associée à cette publication
getWFNoteList()	Retourne la liste des WFNote associées à cette publication
getWState()	Retourne le WFState dans lequel est la publication.
getWFStateLabel(lang)	Retourne le libellé du WFState dans lequel est la publication.
getWFStateLabelHtml(lang)	Retourne le libellé en HTML du WFState dans lequel est la publication.
getWorkflow()	Retourne le workflow de la publication.
hasAlreadyVoted(mbr)	Retourne true si le membre a déjà voté choisi une transition (dans le cas d'un workflow à approbation multiple)
isInRole(roleId, mbr)	Retourne true si le membre indiqué peut utiliser le rôle indiqué.
isInVisibleState()	Retourne true si la publication est dans un état visible du workflow.
isRoleAssigned()	Retourne true si le rôle indiqué est affecté.
unassignMember(mbr)	Retire le membre de tous les rôles associés à cette publication.
unassignRole(roleId)	Retire tous les membres du rôle indiqué.

7.9 Spécialisation des alertes

A partir de JCMS 9 SP1, les alertes de workflows peuvent avoir des messages personnalisés.

La personnalisation passe par la création de propriétés de langues au format indiqué ci-après. L'alerte est alors envoyée en testant séquentiellement les différentes propriétés pour voir si elles sont définies. Chaque propriété peut être utilisée avec les paramètres par défaut des messages de workflow (comme ws, state, reminder ou formName).

Le format des propriétés attendues est :

```
wf-alert.<WorkspaceId>.<Type>.<WorkflowId>.<workflow state>.<transition
type>.<actor>.<reminder>.<propertyName>
```

Les valeurs possibles pour les champs sont :

- WorkspaceId : un identifiant de workspace ou ANY
- Type : le type de publication ou ANY
- WorkflowId : l'id du workflow
- Workflow state : l'état numérique du workflow ou ANY
- transition type : la chaîne de caractères "in", "out", "form-submission" ou ANY
- actor : la chaîne de caractères "alertAuthor", "alertWorkers", "alertAdmins" ou ANY
- reminder : la chaîne de caractères "reminder", "noReminder" ou ANY
- propertyName : la chaîne de caractères "title", "short-description" ou "description"

Par exemple, pour produire une alerte spécifique à l'auteur lors de la sortie de l'état 21 dans le workflow vacationRequestWorkflow, il faut déclarer les propriétés suivantes :

```
wf-alert.ANY.VacationRequest.vacationRequestWorkflow.-21.out.alertAuthor.ANY.title:
Acceptation de {data}

wf-alert.ANY.VacationRequest.vacationRequestWorkflow.-21.out.alertAuthor.ANY.short-
description: Votre demande de congé a été acceptée : {data.url}

wf-alert.ANY.VacationRequest.vacationRequestWorkflow.-21.out.alertAuthor.ANY.description:
<p><a href="{data.url}">Votre demande de congé</a> a été acceptée.</p>
```

7.10 Présentation des libellés des états

A partir de JCMS 9 SP1, les labels des états de workflow comportent un sélecteur CSS qui permet de les identifier sans ambiguïté. Cela permet par exemple d'affecter une couleur spécifique à un état.

Le format du selecteur CSS est :

```
.wfstate.<wfId>-pstatus<pstatus>
```

Par exemple, pour coloriser en vert "success" l'état -21 du workflow vacationRequestWorkflow, on peut écrire en LESS :

```
.wfstate.vacationRequestWorkflow-pstatus-21 {
  background-color: @brand-success;
}
```

8 La gestion des fichiers déposés

8.1 Les différents types de fichiers déposés

8.1.1 Les fichiers des FileDocument

Les fichiers associés aux FileDocument sont déposés dans l'arborescence uploads/docs. Ils sont classés dans des sous-répertoires correspondant à leur type mime puis dans un répertoire représentant l'année et le mois du dépôt. Le nom du fichier reçu est nettoyé : les lettres sont passées en minuscule et désaccentuées, les espaces sont remplacés par des « _ » et les autres caractères non alphanumériques sont ignorés.

Exemple :

Si un contributeur dépose le fichier « Rapport d'étonnement de Jean-Marc.pdf» en juillet 2013, il sera enregistré dans :

```
upload/docs/application/pdf/2013-07/rapport_detonnement_de_jeanmarc.pdf
```

Si le nombre de fichiers au sein d'un répertoire de dépôt dépasse 4096 (propriété channel.max-files-per-dir) alors les nouveaux fichiers sont déposés dans un sous-répertoire.

8.1.2 Les fichiers des photos des membres

Les photos des membres sont stockées de la même manière que les documents mais dans le répertoire `upload/photos/`. Les fichiers satellites sont gérés de la même manière.

8.1.3 Les fichiers des favicons

Une favicon est une petite icône (généralement de 16px) qui représente un site. Elles sont notamment utilisées lors de l'affichage des objets WebPage.

Ces fichiers sont tous stockés dans les répertoires `upload/favicons/`.

8.2 Les fichiers satellites

De nombreux fichiers accompagnent les fichiers déposés dans JCMS :

- Des vignettes générées dans différentes résolutions
- Les fichiers PDF générés pour les fichiers bureautiques (avec le module Convertisseur PDF)
- L'extraction du texte des documents bureautiques (avec le module Indexation des documents)
- Les fichiers de la visionneuse des documents (avec le module Visionneuse de documents)

- …

Les fichiers satellites associés à un fichier déposé sont regroupés dans un sous-répertoire situé dans le même répertoire que le fichier et portant le nom du fichier avec le suffixe `.associated`.

Lorsqu'une nouvelle version d'un document est déposée, l'ensemble des fichiers satellites est effacé et reconstruit.

Lorsqu'un document est supprimé, l'ensemble des fichiers satellites est supprimé.

8.3 La gestion des fichiers des FileDocument

La classe FileIndexManager gère la liaison entre les fichiers physique et les FileDocument. Sa méthode getIndexedDataSet() permet de retrouver rapidement à quelle FileDocument est attaché un fichier physique du répertoire upload/docs/.

Les méthodes channel.getOrphanFiles(…) recherchent l'ensemble des fichiers présents dans le répertoire upload/docs/ et qui ne sont pas rattachés à un FileDocument.

8.4 La génération des vignettes

Afin d'alléger le poids global des pages HTML, JCMS prend en charge la génération des vignettes pour les images. Les vignettes générées sont stockées dans la zone des fichiers satellites.

Quel que soit le format des documents, les vignettes sont produites au format JPEG.

Le tag <jalios:thumbnail/> est fourni par la TLD de JCMS pour insérer rapidement une vignette dans une page JSP.

Exemple :

```
<jalios:thumbnail data="<%= doc %>" width="200" height="150" />
```

A partir de JCMS 9, il est possible de gérer des vignettes carrées même si l'image d'origine est rectangulaire. Le carré est pris dans la zone centrale du rectangle.

Exemple :

```
<jalios:thumbnail data="<%= doc %>" width="100" height="100" square="true" />
```

Tous les types de fichiers ne disposent pas d'un rendu en vignette. Par défaut, JCMS le propose pour les images de types JPG, GIF et PNG, pour les documents Microsoft Office OpenXML et les documents OpenOffice. Cependant, il est possible de proposer des rendus pour d'autres types de documents, en implémentant l'interface `ThumbnailPolicyFilter`.

Exemple :

```
public class MyThumbnailPolicyFilter extends BasicThumbnailPolicyFilter {
  @Override
  public boolean supportsThumbnail(FileDocument doc, boolean jcms) {
    // ...
  }

  @Override
  public boolean createThumbnail(FileDocument doc, File dest, ImageFormat format, int
maxWidth, int maxHeight, String background, boolean done) {
    // ...
```

```
  }

  @Override
  public boolean createThumbnail(FileDocument doc, File dest, ImageFormat format, int
maxWidth, int maxHeight, boolean done) {
    // ...
  }

}
```

8.5 Les permissions

Le dépôt des documents est contrôlé par les permissions. Les permissions permettent de limiter les fichiers déposés selon leurs types et leur poids. Les permissions se déclarent par des propriétés dans le fichier custom.prop.

Ces propriétés sont de la forme :

```
upload.permission.size.<mime-type> : taille maximum autorisée.
```

Où mime-type représente le type de document concerné. Le mime-type `default` peut être utilisé pour déclarer une permission par défaut pour tous les types de documents.

Exemple :

```
# PDF limité 100 MB
upload.permission.size.application/pdf: 100048576

# Images (tout type confondu) limitées à 1 MB
upload.permission.size.image: 1048576

# Sauf les images JPEG limitées à 100 MB
upload.permission.size.image/jpeg: 100048576

# Aucun fichier de plus 1 GB
upload.permission.size.default: 1073741824
```

8.6 Les quotas

En plus des permissions qui contrôlent le poids de chaque document déposé, il est possible de limiter le poids total des fichiers d'un site JCMS ou d'un espace. Pour cela, JCMS propose un système de quota.

Lorsque la consommation disque globale ou celle d'un espace dépasse le seuil, les dépôts sont refusés et l'administrateur reçoit un mail d'alerte. C'est à l'administrateur concerné d'effectuer les nettoyages nécessaires ou de demander une augmentation du quota.

L'ensemble de l'API de contrôle des quotas est pris en charge par la classe QuotaManager.

8.7 Contrôle des dépôts

L'API de JCMS permet d'intervenir avant et après le dépôt d'un fichier.

Ce contrôle se fait en écrivant une classe qui dérive de BasicRightPolicyFilter et surcharge les méthodes suivantes :

```
public boolean checkBeforeUpload(String fieldName, String contentType, String fileName)
public boolean checkAfterUpload(DocUploadInfo info)
```

La méthode checkBeforeUpload() est invoquée au début du transfert et permet de l'interrompre celui en se basant sur le nom du fichier (fileName) ou le type de fichier (contentType).

La méthode checkAfterUpload() est invoquée une fois que le fichier a été déposé. L'attribut info permet d'obtenir des informations sur le fichier. Si cette méthode renvoie true, le fichier est déplacé de la zone temporaire de dépôt à son emplacement définitif. Sinon, le fichier est supprimé et l'utilisateur est prévenu du rejet.

Pour plus de détails, consultez l'article *Personnalisation de la gestion des droits* (http://community.jalios.com/howto/rightpolicy).

9 La gestion des tâches planifiées

JCMS utilise en interne des tâches planifiées pour déclencher des actions à une date donnée. Par exemple, lorsqu'un contenu reçoit une date de publication et passe dans l'état planifié, une alarme est enclenchée pour gérer son passage à l'état publié. Cette gestion d'alarme est également utilisée avec les Worklow Express mais aussi pour envoyer les notifications ou encore pour générer le rapport du journal des accès.

L'API de JCMS vous permet de programmer vos propres tâches planifiées en utilisant le package com.jalios.jdring.

9.1 L'API JDring

JDring est un package qui gère des planifications d'alarmes d'une manière similaire aux commandes cron et at des environnements Unix. Les alarmes peuvent être ajoutées dynamiquement, dans n'importe quel ordre, et peuvent être répétitives ou non. JDring a été conçu pour gérer de grande quantité d'alarmes sans que les performances de JCMS soient dégradées. JDring n'utilise qu'une seule thread qui s'endort jusqu'à la prochaine alarme (via la méthode wait(time)).

L'API de JDring comporte principalement 4 classes :

- AlarmEntry : cette classe contient les paramètres de l'alarme ;
- AlarmListener : cette interface doit être implementée par les objets qui doivent être notifiés lors du déclenchement d'une alarme ;
- TransactionalAlarmListener : cette classe abstraite doit être surchargée par les objets qui doivent être notifiés lors du déclenchement d'une alarme et qui effectuent un traitement en relation avec JcmsDB ;
- AlarmManager : cette classe gère les AlarmEntry. Elle permet d'ajouter et de retirer des alarmes.

Pour gérer une nouvelle alarme, il suffit de créer une AlarmEntry que l'on ajoute à un AlarmManager. L'AlarmEntry doit contenir un AlarmListener qui sera invoqué lorsque la date de l'alarme sera atteinte.

9.1.1 La classe AlarmEntry

La construction d'une AlarmEntry peut se faire via plusieurs constructeurs :

- public AlarmEntry(Date date, AlarmListener listener)
  Construit une alarme pour une date donnée.
- public AlarmEntry(int delay, boolean isRepetitive, AlarmListener listener)
  Construit une alarme (éventuellement répétitive) pour une durée donnée
- public AlarmEntry(int minute, int hour,int dayOfMonth, int month, int dayOfWeek, int year, AlarmListener listener)
  Construit une alarme pour une planification précise et éventuellement répétitive
- public AlarmEntry(String schedule, AlarmListener listener)
  Construit une alarme pour une planification exprimée selon le format (simplifié) de la commande cron:
  [minute] [heure] [jour du mois] [mois] [jour de la semaine] [année].
  Les astérisques (*) spécifient que l'alarme doit être déclenchée de façon répétitive sur cette période.
  Les jours de la semaine commencent par le dimanche et leurs valeurs vont de 1 à 7.
  Exemples :
  30 10 * * * * : tous les jours à 10h30
  30 10 1 * * * : tous les 1er du mois à 10h30
  30 10 * * 2 * : tous les lundi à 10h30

9.1.2 L'interface AlarmListener

Un AlarmListener doit implémenter la méthode handleAlarm() qui est invoquée lorsque l'alarme se déclenche. Elle reçoit en paramètre l'AlarmEntry correspondant.

Exemple :

```
public class MyScheduledTask implements AlarmListener {

  public void handleAlarm(AlarmEntry entry) {
    System.out.println("Wake up !");
  }

}
```

9.1.3 La classe TransactionalAlarmListener

Lorsque le traitement effectué au déclenchement de l'alarme nécessite des interactions avec la base de données JcmsDB, il doit être encapsulé dans une transaction. Cela concerne les traitements effectuant des recherches ou des écritures dans JcmsDB mais aussi les écritures dans JStore qui sont bien souvent accompagnées de requêtes à JcmsDB.

La classe abstraite TransactionalAlarmListener offre cette encapsulation. Il suffit simplement de dériver de cette classe et d'implémenter la méthode handleTransactionalAlarm().

Exemple :

```
public class MyDBScheduledTask extends TransactionalAlarmListener {

  public void handleTransactionalAlarm(AlarmEntry entry) {
    int dataCount = channel.getDataCount(ArchivedPublication.class);
    System.out.println(dataCount + " archive(s) stored in JcmsDB");
  }

}
```

9.1.4 La classe AlarmManager

L'enregistrement des alarmes se fait au près d'un AlarmManager via la méthode addAlarm(). Pour obtenir un AlarmManager, il est recommandé de passer par la méthode channel.getCommonAlarmManager(). Cette méthode gère la majorité des AlarmManager utilisés dans JCMS. Elle garantit en particulier l'arrêt correct de ces AlarmManager (et de leur thread) lorsque le site redémarre.

Il est aussi possible d'utiliser son propre AlarmManager. L'utilisation d'un AlarmManager dédié permet de faire des opérations supplémentaires, comme par exemple la suppression de toutes les alarmes (ce qu'il ne faut surtout pas faire sur le gestionnaire commun). En contrepartie, une nouveau thread est créé. Pour obtenir un AlarmManager, il faut utiliser la méthode channel.getAlarmManager(String key). L'attribut key représente le nom de l'AlarmManager.

9.2 Intégration dans JCMS

9.2.1 Intégration déclarative

La déclaration d'un AlarmListener peut se faire directement dans le fichier plugin.xml du module. Il suffit d'indiquer la classe de l'AlarmListener, la planification de l'alarme avec une syntaxe cron et éventuellement le nom du gestionnaire d'alarme. Si le gestionnaire n'est pas précisé, alors un AlarmManager commun à tous les Plugin est utilisé.

```
<plugin …>
  …
  <plugincomponents>
    <alarmlistener class="com.package.MyAlarmListener"
                   schedule="30 14 * * * *"
                   manager="MyAlarmManager"/>
  </plugincomponents>
  …
</plugin>
```

Alternativement, on peut préciser une fréquence (en minutes) au lieu d'utiliser une syntaxe cron :

```
<plugin …>
  …
  <plugincomponents>
    <alarmlistener class="com.package.MyAlarmListener"
                   frequency="30"
                   manager="MyAlarmManager"/>
  </plugincomponents>
  …
</plugin>
```

9.2.2 Intégration programmatique

Parfois la planification ou la fréquence de l'alarme doit être choisi dynamiquement (p. ex. par la lecture d'une propriété). Dans ce cas, l'ajout d'alarmes peut se faire programmatiquement, par exemple en utilisant un ChannelListener. Un ChannelListener effectue des actions au démarrage de JCMS. Il peut donc enclencher une alarme avec une planification donnée. Pour être pris en charge par le gestionnaire de module, le ChannelListener doit aussi implémenter l'interface PluginComponent.

Exemple de code :

```
public class MyChannelListener extends ChannelListener implements PluginComponent {

  public boolean init(Plugin plugin) {
    return true;
```

```
  }

  public void initAfterStoreLoad() {
    try {
      String schedule = Channel.getChannel().getProperty("myplugin.alarm.schedule");
      AlarmListener alarmListener = new MyAlarmListener();
      AlarmEntry alarmEntry = new AlarmEntry(schedule, alarmListener);
      AlarmManager alarmMgr = Channel.getChannel().getCommonAlarmManager();
      alarmMgr.addAlarm(alarmEntry);
    }
    catch(com.jalios.jdring.PastDateException ex) {
      ex.printStackTrace();
    }
  }
  public void handleFinalize() {  }
  public void initBeforeStoreLoad() {  }
}
```

La déclaration du ChannelListener dans le fichier plugin.xml se fait par la balise <channellistener>.

```
<plugincomponents>
  <channellistener class="com.package.MyChannelListener" />
</plugincomponents>
```

9.2.3 Exemple

TBW

https://community.jalios.com/howto/alarm

10 Autres API

10.1 Alertes

JCMS et ses modules proposent de nombreuses fonctions de notification et d'alerte : notification sur publication, validation dans les workflow, planification d'un événement, rappel sur une tâche, dépassement quotas, ... On dénombre ainsi une cinquantaine d'alertes différentes.

Toutes les alertes émises par la plateforme sont gérées de façon homogène et générique.

L'article suivant présente l'API de gestion des alertes.

https://community.jalios.com/howto/alert

10.2 Authentification

JCMS fournit en standard deux mécanismes d'authentification :

1. Authentification à partir de la base des membres JCMS
2. Authentification à partir d'un annuaire LDAP/LDAPS

Cependant certaines architectures nécessitent des mécanismes d'authentification particuliers ; comme par exemple :

1. Base des utilisateurs gérée dans une base de données
2. Base des utilisateurs gérée dans plusieurs annuaires LDAP non synchronisés
3. Authentification unique (Single Sign-On)
4. Authentification à base de certificats
5. Authentification forte (carte à puce, dispositif biométrique, …)

L'API de JCMS permet de développer et d'intégrer très facilement de nouveaux mécanismes d'authentification adaptés aux besoins de l'architecture cible.

L'article suivant présente l'API d'authentification :

https://community.jalios.com/howto/authenticationhandler

10.3 QueryFilter

L'API QueryFilter permet de compléter ces mécanismes standards en agissant sur les paramètres d'une recherche et en modifiant l'ensemble de résultats obtenus par la méthode standard de recherche. L'API permet aussi d'intégrer les QueryFilter aux interfaces de recherche sans avoir à modifier les JSP.

L'article suivant présente l'API des QueryFilter :

https://community.jalios.com/howto/queryfilter

10.4 Log4J

JCMS utilise la librairie log4j pour sa gestion d'évènements.

L'article suivant décrit le paramétrage et l'utilisation de cette API dans JCMS :

https://community.jalios.com/howto/log4j

10.5 Statistiques et analyse des usages

JCMS intègre un système det collecte et d'analyse des usages. Celui-ci produit des métriques sur les différents pans fonctionnels de JCMS :

- Consultation : visite, téléchargement, contenu les plus consultés, …
- Recherche : taux de recherche, mots les plus recherchés, …
- Contribution : évolution du nombre de contenu, de document, types les plus utilisés, …
- Utilisateurs : évolution du nombre d'utilisateurs, répartition selon le type (membre, contact, invités, …)
- Exploitation : évolution de l'espace disque, des espaces de travail, …
- Technique : taux de requête, temps moyen de réponse, navigateurs et système d'exploitation utilisés, …

L'article suivant décrit cette API :

https://community.jalios.com/howto/analytics

10.6 Open API

Les systèmes d'informations d'entreprises reposent sur des parcs aux technologies hétérogènes (J2EE, .NET, PHP, Ruby on rails). Cela rend le besoin d'interopérabilité de plus en plus indispensable. Jusqu'à présent, il a été possible de répondre à ces besoins avec JCMS en effectuant des développements spécifiques côté JCMS (JSP ou servlets), appelés de manière distante (en HTTP) ou l'inverse.

JCMS propose Open API, une API de Services Web native, de manière à :

- offrir une interface indépendante du langage avec lequel l'API est invoquée ;

- industrialiser les développements : faire reposer les développements sur un socle solide et maintenu ;
- pouvoir réaliser des composants réutilisables : par exemple des modules d'intégrations entre plusieurs applications JCMS.

L’article suivant décrit Open API :

https://community.jalios.com/howto/jcmsopenapi

10.7 Tests Unitaires

Afin de vous assurer du bon fonctionnement de vos développements Java, il est recommandé de développer des tests unitaires. En réalisant ces tests, vous validez que votre code répond aux spécifications fonctionnelles et vous limitez l'apparition de dysfonctionnements lors de vos futures modifications (tests de non régression).

L’article suivant présente l'utilisation de l'outil de tests unitaires JUnit au sein de JCMS :

https://community.jalios.com/howto/junit